Markt&Technik

Winfried Kassera
Frank Kassera

C64

Programmieren in Maschinensprache

Ein Lehr- und Übungsbuch mit ausgewählten ROM- und RAM-Routinen für die Entwicklung von eigenen Assemblerprogrammen. Alle Beispiele für die 40er- und 80er-Serien verwendbar.

Enthalten:
Diskette mit allen
Beispielen

C 64 – Programmieren in Maschinensprache

Winfried Kassera
Frank Kassera

C 64 – Programmieren in Maschinensprache

Ein Lehr- und Übungsbuch mit ausgewählten ROM- und RAM-Routinen für die Entwicklung von eigenen Assemblerprogrammen

Alle Beispiele für die 40er- und 80er-Serien verwendbar

Markt & Technik Verlag

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Kassera, Winfried:
C 64 — Programmieren in Maschinensprache : e. Lehr- u. Übungsbuch mit ausgew. ROM- u. RAM-Routinen für d. Entwicklung von eigenen Assemblerprogrammen / Winfried Kassera ; Frank Kassera. —
Haar bei München : Markt-und-Technik, 1985.
ISBN 3-89090-168-9
NE: Kassera, Frank:

Die Informationen im vorliegenden Buch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen.
Trotzdem können Fehler nicht vollständig ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.
Die gewerbliche Nutzung der in diesem Buch gezeigten Modelle und Arbeiten ist nicht zulässig.

»Commodore 64« ist eine Produktbezeichnung der Commodore Büromaschinen GmbH, Frankfurt, die ebenso wie der Name »Commodore« Schutzrecht genießt. Der Gebrauch bzw. die Verwendung bedarf der Erlaubnis der Schutzrechtsinhaberin.

15 14 13 12 11 10 9 8 7 6 5 4 3
89 88 87 86

ISBN 3-89090-168-9

© 1985 by Markt & Technik, 8013 Haar bei München
Alle Rechte vorbehalten
Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Jantsch, Günzburg
Printed in Germany

Inhaltsverzeichnis

1 Zur Programmierung in ASSEMBLER 13

1.1 Warum nicht bei BASIC bleiben? 15
1.2 Hinweise zum Gebrauch des Buches 17
1.2.1 Hardware/Software-Voraussetzungen 17
1.2.2 Aufbau der Beispielprogramme 18
1.2.3 Noch ein guter Tip 19

1.3 Schreibweisen - Vereinbarungen 20

2 Bewegte Bildschirmobjekte 23

2.1 Ein Beispiel: "Wendezeigerpinsel", ein Richtungsanzeiger 25
2.1.1 Zielsetzung 25
2.1.2 Anschluß eines Joysticks 26
2.1.3 Aufbau des Programms "01-pinsel" 28
2.1.4 ASSEMBLER-Beispiel für "01-pinsel" 30
2.1.5 Vorbereitungen, Initialisierungen 32
2.1.6 Zusammenstellung der verwendeten Labels 34

2.2 Variieren des Moduls 34
2.2.1 Variable Laufgeschwindigkeit 34
2.2.2 Bildschirmobjekte austauschen 34
2.2.3 Bildschirmgestaltung 35

2.3 Abweichungen für die 40/80XX-Geräte 36
2.4 Erweiterungs- und Übungsaufgaben 37

3 Erweiterung der Interrupt-Routine - ein Beispiel 39

3.1 Die IR-Routine 41
3.2 Erzeugung eines Taktes mit der IR-Routine 42
3.3 Wichtige Adressen des "04-taktmodul" 42
3.4 Ablauf von "04-taktmodul" 43
3.5 ASSEMBLER-Programm "04-taktmodul" 43
3.6 Abstimmung des Taktes 44
3.7 Eine kleine Testroutine zur Bestimmung der Taktfrequenz 44
3.8 Einstellen des IRQ-Vektors 45

4 Zahlen, Variablen - Formate 49

4.1 Rechnen mit Ganzzahlen (Integer-Zahlen) 51
4.1.1 Rechnen mit positiven Ganzzahlen 51
4.1.2 Negative Ganzzahlen 54
4.1.3 Subtrahieren von Integerzahlen im positiven Bereich 56
4.1.4 Subtraktion mit beliebigen Integerzahlen 57
4.1.5 Höhere Rechenarten mit Integerzahlen 57
4.1.6 Integermultiplikation mit INTMUL 60

4.2 Arbeiten mit reellen Zahlen 61
4.2.1 Formate für reelle Zahlen 61
4.2.2 Übernehmen von gespeicherten Zahlen mit MEMFAC 63
4.2.3 Erzeugung von reellen Zahlen 63

4.3 Zahlenumwandlungen 65
4.3.1 Integer- in Realzahlformat mit INTFLP 65
4.3.2 Reelle Zahl in Integerzahl mit FLPINT 65
4.3.3 Umwandlung eines Strings in eine reelle Zahl mit STRFAC ... 66
4.3.4 Umwandlung einer Zahl in einen String mit FLPSTR 67
4.3.5 (A)-Inhalt in ASC-Code mit BYTHEX 68
4.3.6 ASC-Code in Byte umwandeln mit HEXBYT 68
4.3.7 Positive Integerzahl in Realzahl mit ADRFLP 68

Umwandlungs-ROM-Routinen (Zusammenstellung) 69

5 Arithmetik mit ROM-Routinen 71

5.1 Durch 10 dividieren mit FDIV10 74
5.2 Mit 10 multiplizieren 75
5.3 Addieren des Werts 0.5 mit ADD0.5 75
5.4 Addieren beliebiger Zahlen mit ADD 76
5.5 Addieren beliebiger Zahlen mit M-ADD 76
5.6 Subtrahieren mit M-SUB 77
5.7 Vorzeichenwechsel mit FACMIN 78
5.8 Betrag einer Zahl mit FACABS 79
5.9 Multiplizieren mit M-MULT 79
5.10 Division mit M-DIV 80
5.11 Kehrwert bilden mit M-DIV 80
5.12 Quadratwurzel ziehen mit SQRFAC 81
5.13 Potenzieren und Radizieren mit POTRAD 81
5.14 Logarithmieren mit LOGNAT 83
5.15 Exponentialrechnen mit EHOCHF 83

5.16 Erzeugen einer Zufallszahl mit ZUFALL 84
5.17 Winkelfunktionen mit SINUS, COSIN, TANG 85
5.18 Umkehrung der Winkelfunktionen mit ARCTAN 87
5.19 Weitere Arcusfunktionen mit ARCTAN 88
5.20 Polynomauswertung mit POLNOM 89
5.21 Wertetabellen für Funktionen mit POLNOM 91
5.22 Vergleichsoperationen mit CMPFAC 94
5.23 Vorzeichenprüfung mit SGNFAC 96

Arithmetik-ROM-Routinen (Zusammenstellung) 98

6 Bildschirmoperationen 101

6.1 Ausgabe einer Integerzahl mit INTOUT 103
6.2 Ausgabe einer reellen Zahl mit FLPOUT 103
6.3 Ausgabe eines ASCII-Zeichens mit CHROUT (BSOUT) 104
6.4 Vorbereitete Zeichenausgaben 105

6.5 Cursorposition festlegen 105
6.5.1 Cursorposition C64 105
6.5.2 Cursorposition 40/80XX 106

6.6 Ausgabe eines Strings mit STROUT 109
6.7 Umwandlung des FAC-Inhalts in einen String mit FACSTR 109

6.8 Anwendung: eine PRINT USING-Routine 111
6.8.1 Ablauf-Struktur der PRINT USING-Routine 113
6.8.2 Struktogramm zur PRINT USING-Routine 118
6.8.3 ASSEMBLER-Modul "5l-printusing" 119

6.9 Ausgabe von Hexzahlen 123
6.9.1 Byte in der Hexform ausgeben mit BYTOUT 123
6.9.2 Vierstellige Hexzahl (Adresse) ausgeben mit ADROUT 123
6.9.3 Zwei Zeichen ausgeben mit OUT2 124

6.10 Bewegungssimulation - eine Kompaßanzeige 124

Ausgabe-Routinen (Zusammenstellung) 133

7 Eingabe-ROM-Routinen 135

7.1 Eingabe eines Zeichens über die Tastatur mit GETIN 137
7.2 Künstlicher Cursor mit GETIN und CHROUT 138

7.3 Zahleneingabe (reell) mit GETIN und STRFAC 141
7.4 Eingabe mit BASIN 143
7.5 Eingabe einer Zeile mit INLINE 145
7.6 Eingabe von Hexzahlen mit HEXINB und HEXINA 148

Eingabe-ROM-Routinen (Zusammenstellung) 149

8 Verwaltung der Variablen 151

8.1 Überblick über die BASIC-Variablen 153
8.1.1 Lage der Variablen im RAM 153
8.1.2 Variablen-Arten 154
8.1.3 Struktur der Variablen 154

8.2 Einrichten einer Variablen 154
8.2.1 Festlegen des Bereichsanfangs 156
8.2.2 Suchen oder Einrichten einer Variablen mit PTRVAR 158

9 Bedienung von Peripherie 161

9.1 Datentransfer über den IEC- bzw. den seriellen Bus 163

9.2 Umschaltungen des seriellen bzw. des (IEC)-Bus 163
9.2.1 Datenübernahme mit TALK 165
9.2.2 Datenausgabe mit LISTEN 166
9.2.3 Beispiel: "druckerausgabe" mit LISTEN, BSOUT, CLALL 166
9.2.4 Modul "druckex": Drucker als Schreibmaschine 168
9.2.5 Vorbereitung von Datenübertragungen mit OPEN 171
9.2.6 Ausgabevorbereitung mit CHKOUT 171
9.2.7 Eingabevorbereitung mit CHKIN 172
9.2.8 Standard-Ein/Ausgabe herstellen mit CLRCH 172
9.2.9 Dateien schließen mit CLALL 172
9.2.10 Schließen einer Datei mit CLOSEA und CLOSEL 173

9.3 Vereinfachungen zur Dateibehandlung 173
9.4 Behandlung von Dateien - Beispiele 174

9.5 Arbeiten mit SEQ-Dateien 175
9.5.1 Öffnen einer Datei 175
9.5.2 Beispiel: Schreiben mit CHKOUT und BSOUT 177
9.5.3 Beispiel: SEQ-Lesen mit CHKIN und BASIN 178
9.5.4 Beispiel: Schließen der Datei mit CLOSEA und CLOSEL 179

9.5.5 Verknüpfen der SEQ-Routinen 180

9.6 REL-Dateien - Schreiben/Lesen mit OUTBUS/INBUS 181
9.6.1 REL-Dateien auf dem C64 181
9.6.2 REL-Dateien auf 4Oer- und 8Oer-Geräten 190

9.7 Laden eines Programmes mit LOAD und LOADXX 196

9.8 Anwendung: Modul zum Nachladen von Programmen 197
9.8.1 Das Maschinenprogramm "81-loadmodul" 198
9.8.2 BASIC-Hilfsprogramm zur Übernahme von Maschinenteilen 199

9.9 Modul "84-quickdirector" mit TALK, INBUS, UNTALK 202
9.10 Modul "85-printdirector" (Floppy-Drucker) 208
9.11 Direktzugriffe auf Floppy: Modul "86-fastdisk" 216

Adressen und ROM-Routinen zur Ein/Ausgabe 218

10 Maschinenmodule in BASIC-Programmen 223

10.1 Übernahme von BASIC-Parametern 225
10.2 Zeichen aus dem BASIC-Text holen mit CHRGET und CHRGOT 226
10.3 Byte-Auswertung mit GETBYT und VALBYT 228
10.4 Eine Anwendung: PRINT AT-Routine mit Fehlermeldung 233
10.5 Zu den Fehlermeldungen 235
10.6 Zur Schreibweise der BASIC-Befehle 236
10.7 Auswertung mit VAREAL 238
10.8 Auswertung mit VALKLA 239
10.9 Auswertung von Integerzahlen mit VALINT und INTADR 239
10.10 Auswertung mit VALPAR, VALSTR, PARFLG und TYPFLG 241
10.11 Übernahme einer BASIC-Variablen mit GETVAR 243
10.12 Ein vielseitiges Modul: "98-onstring" 244
10.13 "99-onstring" für 4O/8OXX-Geräte 250
10.14 Verknüpfen von Modulen - zwei Anwendungen 254
10.15 Modulverknüpfung mit einer Sprungleiste 258

BASIC-Text-Routinen und -adressen (Zusammenstellung) 261

11 Diverse ROM-Hilfen - Anwendungen 265

11.1 BASIC-Start vom Maschinenprogramm aus mit RUN 267
11.2 Warmstart mit MGOTO ab einer bestimmten Zeilennummer 267
11.3 Startadresse einer BASIC-Zeile suchen mit BLINAD 270
11.4 Umschalten von Text- auf Graphikmodus 270
11.5 Abfrage der STOP-Taste mit STOPRY und STOPO 272
11.6 Sprung in den READY-Modus mit MREADY 272
11.7 Verschieben von RAM-Bereichen mit TRABLO 272
11.8 Abspeichern eines Datenbereichs mit MSAVE 274
11.9 Laden eines Programms mit veränderter Startadresse 275
11.10 Speichern eines Datenblocks mit variabler Anfangsadresse ...279

12 ASSEMBLER-Kurzschule 283

12.1 Die Register des 65XX-Prozessors 285
12.2 Das Prozessor Statusregister P 285
12.3 Beeinflussung der Flags durch Befehle 285

12.4 Der Befehlssatz in ASSEMBLER 286
12.4.1 Eingabebefehle (Ladebefehle) 287
12.4.2 Ausgabebefehle (Speicherbefehle) 288
12.4.3 Arithmetische Verknüpfungen 289
12.4.4 Logische (bitweise) Verknüpfungen 291
12.4.5 Verschiebebefehle (bitweise) 292
12.4.6 Vergleichsbefehle 293
12.4.7 Transportbefehle zwischen den Registern 294
12.4.8 Sprungbefehle 295
12.4.9 Beeinflussung der Flags des Statusregisters 296
12.4.10 Lückenfüller 296

13 ROM-Routinen - thematisch, mit Kurzbeschreibung 297

14 ROM-Routinen - alphabetisch 313

15 Wichtige Adressen - alphabetisch 319

16 Stichwortverzeichnis 325

17 Übersicht weiterer Markt&Technik-Bücher..... 328

Vorwort

Dieses Buch ist zunächst entstanden als Sammlung von Programmierschritten in ASSEMBLER, die sich bei der Lösung von Simulationsproblemen, Bildschirmgestaltung, Textverarbeitung usw. auf COMMODORE-Geräten ergeben hatten. Wir haben dann den Themenkreis erweitert und so gut wie alle Punkte erfaßt, die für Maschinenprogramme relevant sind.

Als Schwerpunkt hat sich dabei der Einsatz der betriebsinternen ROM-Routinen herauskristallisiert, der die Erstellung von ASSEMBLER-Programmen wesentlich vereinfacht. Denn was es schon gibt, braucht man nicht noch einmal erfinden.

Wir sagen Ihnen aber nicht nur, wo die einzelnen Einsprungadressen liegen (das tun viele Veröffentlichungen), sondern erklären auch die Zusammenhänge anhand von sehr einfachen bis sehr anspruchsvollen Beispielen. Unser Prinzip heißt hier: Vom Einfachen zum Schweren. Und damit ergibt sich ein methodisch sinnvoller und für den Leser leicht nachvollziehbarer Aufbau.

Alle vorkommenden Routinen wurden mehrfach von Frank Kassera getestet und verbessert, der zu diesem Buch auch eine Diskette für den C64 mit allen beschriebenen Programmen zusammengestellt hat.

Trotz aller Sorgfalt und Mühe, die wir uns beim Schreiben, Testen und Korrigieren gegeben haben, ist es nicht ganz auszuschließen, daß sich in den Text der eine oder andere Tipp- oder Druckfehler eingeschlichen hat, der uns trotz mehrfacher Durchsicht entgangen ist. Wir versichern Ihnen aber, daß alle vorgestellten Programme einwandfrei laufen: Nicht ein einziges wurde nur so "auf Papier" entwickelt. Sollten Sie daher auf Unstimmigkeiten stoßen, dann könnte eventuell die Unbarmherzigkeit des Druckfehlerteufels dahinterstecken. Für einen entsprechenden Hinweis - vielleicht schon mit einem Verbesserungsvorschlag - wären wir Ihnen dankbar.

Winfried Kassera, Frank Kassera

1

Zur Programmierung in ASSEMBLER

1 Zur Programmierung in Assembler

1.1 Warum nicht bei BASIC bleiben?

Das COMMODORE-BASIC ist für den C64 und bis zu den 8OXX-Geräten nicht gerade üppig ausgestattet. Trotzdem läßt sich diese Computersprache relativ einfach lernen und anwenden. Aber bald wird der BASIC-Programmierer feststellen, daß er etliche Ideen nicht mehr verwirklichen kann, wenn es darum geht, möglichst rasche Abläufe zu erzeugen. Vor allem bei der Erstellung von Spielen und bei der Arbeit mit Dateien läuft so manches entnervend langsam ab.
Spezielle Programmiersprachen für bestimmte Zwecke schaffen zwar Erleichterungen, können aber neue Probleme aufwerfen.

Wer also gleich "Nägel mit Köpfen" machen will, der stürzt sich in das Abenteuer Maschinensprache und hat damit die schnellste aller Computersprachen gewählt. Wem es dann noch zu langsam geht, der braucht einen neuen Computer mit einem schnelleren Mikroprozessor.

Maschinensprache besteht eigentlich nur aus Zahlen, genauer gesagt aus Speicherinhalten in Byteform, wobei jeder möglichen Zahl ein Befehl zugeordnet ist, der wieder darüber entscheidet, ob die nächste Zahl ein Befehl oder eine Speicherstelle o.ä. ist.

Damit man nicht umständlich mit diesen sog. OP-Codes operieren muß, setzt man ASSEMBLER-Programme ein, die praktisch eine indirekte Programmierung in Maschinensprache erlauben, indem sie die Befehle in abgekürzter Form als Buchstabenfolge annehmen und in den entsprechenden Code verwandeln.

Mit einem Disassembler-Programm kehrt man das Ganze wieder um: Die Zahlenfolge eines Maschinenprogramms wird in abgekürzten Befehlen dargestellt. Die Abkürzungen lassen durch ihre Zeichenfolge den entsprechenden Befehl erkennen. Man bezeichnet sie mit dem zungenbrecherischen Begriff "Mnemonics".

Um die folgenden Kapitel zu verstehen, sollten Sie daher neben ordentlichen BASIC-Kenntnissen auch schon einmal ein paar erfolgreiche Versuche in ASSEMBLER unternommen haben. Wenn nicht,

dann können Sie zwar so nach und nach durch die einzelnen Kapitel hindurch Ihre Kenntnisse erweitern, werden aber an einigen Stellen hart kämpfen müssen.

Vorsichtshalber finden Sie am Schluß des Buches in Kapitel 12 eine kurze, komprimierte ASSEMBLER-Schule, die Ihnen immer dann weiterhelfen soll, wenn Sie mit den ASSEMBLER-Befehlen und vor allem mit den verschiedenen Adressierungen Schwierigkeiten bekommen sollten.

Sollten Sie jedoch noch nicht in der Lage sein, Bits und Bytes zu unterscheiden, dann empfehlen wir Ihnen, dieses Buch noch etwas zurückzulegen und sich erst einmal gründlich mit dem bisher vernachlässigten Teil Ihres Computer-Handbuchs vertraut zu machen.

Wenn Sie aber schon ein As in BASIC sind, kommt Ihnen Ihr dort erworbenes Wissen sicher auch bei unseren Themen sehr zugute. Ihr BASIC-Wissen sollten Sie aber trotzdem weiter ausbauen, denn die meisten guten Programme bestehen aus einer gesunden Mischung zwischen einer höheren Sprache (BASIC ist eine) und Maschinenteilen, die immer dort eingesetzt werden, wo schnell und oft gearbeitet werden muß.

Oft besteht das Rahmenprogramm aus BASIC und der harte Kern läuft "in Maschine". Oder aber es werden von BASIC aus immer wieder die notwendigen Module aufgerufen.
Wir werden uns diese Möglichkeiten im Laufe des Textes anschauen.

Zusammengefaßt: Wir werden die Verbindung zu BASIC auf keinen Fall abreißen lassen. Im Gegenteil: Unsere in ASSEMBLER geschriebenen Maschinenprogramme werden einerseits das COMMODORE-BASIC unterstützen und verbessern, andererseits werden die bereits vorhandenen Routinen in Ihren ROMs unsere ASSEMBLER-Programmierung erleichtern.

Sie werden sehen, es bereitet nicht nur Mühe, sich durch die einzelnen Kapitel durchzunagen, sondern es macht auch Spaß, mitten in den ROM-Topf zu greifen und die herausgefischten Nüsse zu knacken.

1.2 Hinweise zum Gebrauch des Buches

1.2.1 Hardware/Software-Voraussetzungen

Alle Programme, die Sie in den folgenden Kapiteln finden, sind zunächst für den normalen COMMODORE C64 geschrieben. Mit nur wenigen Abänderungen laufen sie fast alle auch auf den 40/80XX-Geräten. Sie benötigen dazu keinerlei Erweiterungen oder Ergänzungen.

Was Sie haben sollten ist also:
- das normale Grundgerät (C64,40XX,80XX)
- eine Floppystation (1541 bzw. 4040/8050 o.ä.)
- evtl. einen Joystick
- evtl. einen Drucker

Wir können nicht auf alle Spezialitäten des C64 eingehen. Doch mit dem hier erworbenen Grundwissen sind Sie in der Lage, auch Sound und Graphik in ASSEMBLER zu programmieren, da die PEEK- und POKE-Befehle aus BASIC für die Umsetzung in ASSEMBLER wohl kaum mehr Schwierigkeiten bereiten dürften. Eine Unterstützung durch ROM-Routinen gibt es hier ohnehin (leider) nicht.

An Software benötigen Sie auf jeden Fall ein Assembler/Disassembler-Programm. Alle Ausführungen sind so gehalten, daß Sie auch mit den einfachsten Versionen arbeiten können. Komfortabel wäre natürlich ein Editor, der Ihre Labels aufnimmt und assembliert. Notwendig ist er aber nicht.

Es wäre empfehlenswert, wenn Sie ein kommentiertes ROM-Listing für Ihr Gerät auftreiben könnten. Wir haben Ihnen zwar die Mühe abgenommen, die einzelnen - auch weniger bekannten - Routinen herauszusuchen und zu analysieren. Doch wenn Sie selbst nachvollziehen wollen, was dort geboten wird, sollten Sie dort nachschlagen.

Zumindest setzen wir voraus, daß Sie das entsprechende Handbuch zu Ihrem Rechner besitzen, wo z.B. die Tabellen für die jeweiligen Codes der ASCII-Zeichen, der Tastatur oder der Bildschirmzeichen stehen.

1.2.2 Aufbau der Beispiel-Programme

Wir haben uns lange überlegt, ob wir die Adressen als Dezimalzahlen oder als Hexzahlen schreiben sollen und haben uns für die ersteren entschieden, weil wir glauben, daß manchem Umsteiger die Hexzahlen Probleme bereiten. (Über kurz oder lang werden Sie sich aber trotzdem daran gewöhnen müssen.)

Wem das nicht gefällt, der findet in den Zusammenstellungen der Routinen und Adressen alle Angaben in beiden Zahlensystemen.

An dieser Stelle sei aber doch kurz auf die Hexzahlen eingegangen, da die Dezimalschreibweise eben auch Nachteile hat:

Nehmen wir an, die Zahl 17000 wird in der Form LO/HI benötigt. Dann erscheint sie in Dezimalschreibweise als 104/66, denn es gilt, daß das HI-Byte den 256-fachen Wert des LO-Bytes hat:

104 + 66 mal 256 = 17000

In Hexschreibweise wäre das einfacher, denn 17000 heißt dort $4268. Zerlegen wir in LO/HI, dann ergibt sich ganz einfach: LO=$68 und HI=$42.

Alle Programme sind mit entsprechenden Kommentaren versehen, die es Ihnen ermöglichen, solche Zerlegungen nachzuvollziehen, wenn Sie darin noch nicht so fit sind.

Dabei haben wir uns an folgendes Prinzip gehalten:

- Die ersten Programme sind am ausführlichsten gegliedert und beschrieben.

- Die wichtigsten Programme - also die, an denen man viel lernen kann - gehen auch auf Einzelheiten ein, wenn sie weiter hinten im Buch stehen.

- Wichtige ASSEMBLER-Befehle, von denen wir wissen, daß sie nicht gern von Anfängern verwendet werden, werden extra erläutert.

- Fast alle Programme haben Modulcharakter. Das bedeutet, sie laufen in jedem RAM-Bereich, ohne daß irgendwelche Adressen verändert werden müssen. Wir gehen später näher darauf ein.

- Wir verwenden z.T. LABELs, die nicht der COMMODORE-Schreibweise entsprechen, wenn Sie durch deutsche Abkürzungen verständlicher wirken.

- Wichtige Stellen, Adressen, Labels usw. werden an Ort und Stelle noch einmal wiederholt, so daß Sie nicht lange zu blättern brauchen. Wenn auf umfangreichere Zusammenhänge nicht verzichtet werden kann, finden Sie einen Hinweis zum entsprechenden Abschnitt.

Wenn Sie nicht nur nachvollziehen wollen, was wir Ihnen hier bieten, dann sollten Sie eigene Ideen zur Problemlösung entwickeln. Meist gibt es eine Fülle von Möglichkeiten.
Wir haben nicht immer die beste und eleganteste ausgesucht, sondern haben mehr Wert auf eine leichte Verständlichkeit der Sachverhalte gelegt.

Wenn Sie erst einmal so weit sind, daß Sie bei dem einen oder anderen Programmvorschlag sagen können "Mensch, das ginge ja noch viel einfacher", dann haben Sie einen Riesenschritt in die richtige Richtung getan. Aber, aber, aber: Probieren Sie bitte auch Ihre Version aus, bevor Sie sich als Meister fühlen. Denn oft genug hat man sich zu früh gefreut und die anscheinend bessere Lösungsmöglichkeit funktioniert nicht.

Sie brauchen sich aber nicht zu grämen, uns ist es bei der Erstellung dieses Manuskripts genauso ergangen.
Meist sind es die Randbedingungen, die einem ganz schön Kopfzerbrechen bereiten können. Das Hauptproblem hat in den meisten Fällen einen einfachen Algorithmus (Verfahren zur Lösung logischer Probleme).

1.2.3 Noch ein guter Tip

Maschinenprogramme haben die Eigenart, daß sie keine Fehler vertragen. Bemühen Sie sich daher von vornherein um eine saubere und konzentrierte Arbeitsweise. Es macht wenig Spaß, wenn ein Programm nicht läuft, nur weil man z.B. 17268 statt 17286 eingetippt hat.
Tückischerweise verabschiedet sich der Computer klammheimlich und ist in vielen Fällen nur noch durch Aus- und Einschalten wieder zum Leben zu erwecken. Unser Programm hat er natürlich vergessen.

Beherzigen Sie daher folgenden Rat:

Jedes Programm wird zuerst abgespeichert und dann ausprobiert!

Doch nun genug der Vorreden, tauchen wir ein in das Innenleben unseres elektronischen Sklaven. Aber passen Sie auf, daß er nicht anfängt, Sie zu terrorisieren!!!

1.3 Schreibweisen - Vereinbarungen

Für den Text und die Programmbeschreibungen legen wir folgende Vereinbarungen fest:

- Ein Label ist immer eine Adresse im ROM oder RAM des Adressbereichs. Wir drucken es fett in der Form **LABEL**.

  Beispiel: jsr **FLPOUT**
  Bedeutung: Sprung zur Adresse (43708), mit **FLPOUT**=43708

- Numerische Adressangaben schreiben wir in Klammern, um sie von Zahlen, Werten usw. eindeutig zu unterscheiden, sofern sie außerhalb von ASSEMBLER-Programmen stehen.

  Beispiel: ... und springen nach (17800) ...
  Bedeutung: Fortführung des Programms bei Adresse (17800)

- Inhalte von Adressen, Adressbereichen, Registern usw. schreiben wir in eckige Klammern.

  1. Beispiel: <FAC>=20
  Bedeutung: Wert des FAC ist 20

  2. Beispiel: <A>=255
  Bedeutung: Inhalt des (A)-Registers ist 255

- Einfache Pfeile in Programmen bedeuten eine Wenn-Dann-Folge. Doppelpfeile bedeuten: ... dann Sprung nach ...

```
Beispiel:  cmp #80     ; vergleiche (A) mit 80
           beq 10000   ; richtig ===> Sprung nach (10000)
           rts         ; falsch ---> Ende
```

- Zerlegungen von Adressen erfolgen immer in der Reihenfolge LO/HI, wenn nicht ausdrücklich etwas anderes angegeben ist.

  Beispiel: 18000=80/70=$50/46
  Berechnung: 80+(70mal256)=80+17920=18000

- Bei Hexzahlen stellen wir das Zeichen '$' nur einmal voran.

  Beispiel: <$20/21>
  Bedeutung: Inhalt der Adressen ($20)=LO und ($21)=HI

- Alle Programmbeispiele sind fortlaufend durchnummeriert. Der Programmname steht in Anführungszeichen.

Achtung:
Alle Programmbeispiele für den C64 wurden für die neue Version des Betriebssystems geschrieben, bei der das zusätzliche Belegen des Bildschirm-Farbspeichers nicht mehr notwendig ist, wenn direkt in den Bildschirmbereich geschrieben wird.

Beispiel für die neue Version:

```
      lda #42      ; Code für Zeichen "*"
      sta 1224     ; Ausgabe auf Bildschirmadresse (1224),
                     also auf die 200. Stelle (1024+200)
```

Die alte Version erfordert folgendes Vorgehen:

```
      lda #43      ; Zeichencode
      sta 1224     ; Ausgabe auf Bildschirmadresse (1224)
      ldx #2       ; Farbe Nr.2 (Beispiel für ROT)
      stx 55496    ; in Farbspeicher (55296+200)
```

Anmerkung: Bereich des Farbspeichers: (55296) bis (56295)

2

Bewegte Bildschirmobjekte

2 Bewegte Bildschirmobjekte

2.1 Ein Beispiel: Wendezeigerpinsel

Bei Computerspielen oder Simulationsprogrammen ist es in vielen Fällen erforderlich, daß man über den Bildschirm Figuren oder Gegenstände wie Flugzeuge, Fahrzeuge, Zeiger o.ä. laufen läßt. Die Steuerung dieser Objekte erfolgt dabei entweder über die Tastatur (meist über den numerischen Teil) oder aber über einen Joystick.

2.1.1 Zielsetzung

a) Ein Quadrat soll über den Joystick nach links und nach rechts gesteuert werden können.
b) Die Ausschläge dieses Quadrats sollen rechts und links begrenzt werden.
c) Die Geschwindigkeit der Wanderung der Rechteckfigur soll beeinflußt werden können.
d) Die Stellung des Quadrats muß jederzeit abfragbar sein.

Anwendungsmöglichkeit:
z.B. Anzeigegerät für die Drehbewegung eines Flugzeugs (sog. Wendezeiger). Je größer der Ausschlag, desto höher die Drehgeschwindigkeit.
Das bewegliche Quadrat stellt in diesem Fall einen Zeiger (den sog. Pinsel) dar.

Art und Lage des Programms:
reines Maschinenprogramm beginnend bei Adresse 18000.
(Die Anfangszahl ist willkürlich gewählt.)

Wichtige Adressen und ihre Bedeutung

Label	C64	40/80XX
KEY	203=$cb	151=$97 (SK:155=$9b)
	Code der momentan gedrückten Taste	
	<203>=64	<151>=255
	---> keine Taste gedrückt	

DDRA	56322=$dc02 Datenrichtungsregister	59459=$e843
PADAT	56320=$dc00 (C-Port 2) 56321=$dc01 (C-Port 1) Adressen zur Joystickabfrage	59471=$e8f4

freie Zeropage-Adressen (nicht bei Kassettenoperationen!):

	163...180=$a3...b4	177...195=b1$...c3
Bandpuffer	828...1019= $033c...03f7	826...1017= $033a...03f9

2.1.2 Anschluß eines Joysticks

Beim C64 verwenden wir den Controll-Port 2 zum Anschluß eines Joysticks (ohne Verzögerung).
Um den Joystick abzufragen, wird das Datenrichtungsregister **DDRA** auf "Empfang" geschaltet. Das geschieht dadurch, daß wir diese Adresse mit 0 belegen: <DDRA>=0. Nach dem Einschalten ist dies ohnehin der Fall, so daß wir uns diesen Befehl eigentlich sparen können.

Die Abfrage nach der Stellung des Joysticks erfolgt über die Datenadresse des Ports **PADAT**. Beim C64 ist dies (56320)= ($dc00) bei Port 2.

Die Joysticks sind so gebaut, daß sie immer einen oder zwei Pins des Control-Ports mit Masse kurzschließen und damit die Datenleitungen auf LO legen.
Vom C-Port 2 werden die ersten 7 Bits des Datenregisters beeinflußt. Erfolgt kein Ausschlag, dann stehen alle Bits auf 1:
<56320> = 01111111_2 = 127. Das heißt also, die Adresse (56320) wird mit Byte 127 belegt.

Schauen wir uns die anderen Joystickausschläge der Vollständigkeit halber an (Standard-Joystick):

OBEN (Nord):	01111110 = 126	(Bit 0	gelöscht)
RE/O (Nordost):	01110110 = 118	(Bit 0 und 3	gelöscht)
RECHTS (Ost):	01110111 = 119	(Bit 3	gelöscht)
RE/UN (Südost):	01110101 = 117	(Bit 1 und 3	gelöscht)
UNTEN (Süd):	01111101 = 125	(Bit 1	gelöscht)

```
LI/UN (Südwest):    01111001 = 121  (Bit 1 und 2 gelöscht)
LINKS (West):       01111011 = 123  (Bit       2 gelöscht)
LI/O  (Nordwest):   01111010 = 122  (Bit 0 und 2 gelöscht)
Feuerknopf:         01101111 = 111  (Bit 4       gelöscht)
```

Selbstverständlich sind auch andere Knüppelversionen mit abgewandelten Schaltungen möglich.
Mit einem einfachen Test lassen sich im Zweifelsfall die Joystickeingaben überprüfen:

```
100 ' joystickabfrage port 2 / c64
200 print peek(56320),:goto 200
```

Nach dem Programmstart mit RUN bewegt man den Joystick in die gewünschte Richtung und erhält die Belegung des Datenregisters.

Wir können jetzt brav die Joystickeingabe mit den einzelnen Byte-Werten vergleichen, aber es geht auch schneller:
In unserem Fall erkennen wir, daß bei jeder Rechstbewegung auf jeden Fall das Bit 3, bei jeder Linksbewegung das Bit 2 auf Null steht.
Das vereinfacht uns später die Abfrage nach "rechts" oder "links", wenn wir die Bit-Operationen "AND" oder "BIT" verwenden. An dieser Stelle gehen wir deshalb etwas näher auf diese Verknüpfungen ein:
Finden wir in (56320) einen Wert vor, dann verknüpfen wir ihn mit AND #8, um ihn zunächst auf "rechts" zu überprüfen. Das folgende Beispiel nimmt an, daß 121 (li/un) aufgenommen wurde.
In Bit-Schreibweise sieht das so aus:

```
              01111001    (=121)        lda 56320
       and    00001000    (=  8)        and #8
Ergebnis:     00001000    (=  8)
```

(Bei der bitweisen Operation AND wird nur dann ein Bit auf 1 gesetzt, wenn beide Operatoren an dieser Stelle auf 1 stehen.)

Überprüfen wir nun das Z-Flag, das immer dann gesetzt wird, wenn die letzte Operation mit Null endete, dann ist es in unserem Fall nicht gesetzt. Dies können wir zur Folgerung nehmen:

    Z-Flag nicht gesetzt ---> kein Rechtsausschlag

Anschließend erfolgt in gleicher Weise die Überprüfung auf links:

```
              01111001    (=121)        lda 56320
      and     00000100    (=  4)        and #4
Ergebnis:     00000000    (=  0)
```

Daraus folgt: Z-Flag gesetzt ---> Ausschlag nach links

Bei gesetztem Z-Flag können wir mit BEQ demnach zur entsprechenden Routine verzweigen, da die geforderte Bedingung (Ausschlag nach links) vorliegt. Mit BNE springen wir zur nächsten Überprüfung oder zum Schleifenende, um auf eine Eingabe zu warten, die einen Rechnervorgang erfordert.
Alle diese Operationen werden im (A)-Register durchgeführt.
Mit AND ändert sich dadurch auch der Akku-Inhalt, so daß er für den nächsten Vergleich nicht mehr brauchbar ist.

Die Verknüpfung BIT hat diesen Nachteil nicht: Sie beeinflußt nur die Flags, nicht aber den (A)-Inhalt.

Im letzten Beispiel sieht das so aus:

```
             01111001    (=121)        lda 56320
      bit    00000100    (=  4)     !! bit #4 !!
Ergebnis:    01111001    (=121)   und  Z-Flag = 1
```

Aber Achtung: Den Befehl BIT #4 gibt es in ASSEMBLER nicht, weil BIT keine unmittelbare Adressierung zuläßt. Wir behelfen uns damit, daß wir eine Adresse (in unserem Beispiel (180)) mit dem Wert 4 belegen und bei der Überprüfung mit BIT 180 arbeiten. Die einmalige Belegung von (180) führen wir nicht im Hauptprogramm durch, sondern in einem zugehörigen Vorspann. Doch davon weiter unten.

Der (A)-Inhalt kann also ohne nochmaliges Laden sofort zur nächsten Überprüfung ohne Einschränkung verwendet werden.
Soweit unser kurzer Ausflug in das ASSEMBLER-Wissen. Nun wieder zurück zu unserem Problem.

2.1.3 Aufbau des Programms "Ol-pinsel"

Bevor wir ans Programmieren gehen, sollten wir uns über den Ablauf des Programms einige Vorüberlegungen notieren und die einzelnen Teile wenigstens grob strukturieren.
Noch einmal zur Wiederholung: Unser Ziel ist es einen Bildschirmfleck gesteuert horizontal zu bewegen.

Beim C64 bietet sich dazu ein Sprite an, den wir mit Hilfe der x- und y-Koordinaten über den Bildschirm laufen lassen können.
Es kann nicht Sinn dieses Büchleins sein, die ganzen Graphik-

Raffinessen des C64 zu untersuchen. Aber die wesentlichen Punkte wollen wir ansprechen:

- Das Sprite muß definiert werden. Wir müssen uns auf eine Nummer einigen und eine Farbe festlegen.
  Das geschieht in einem Vorspann zum Hauptprogramm.
- Das Sprite braucht eine Startposition und einen linken und rechten Anschlag.
- Außerdem kann das "Pinsel"-Sprite bei jedem Joystick-Ausschlag in mehr oder weniger großen Sprüngen über den Bildschirm geführt werden. Die feinste Unterteilung ist dabei der einzelne Pixelabstand, der fast eine fließende Bewegung erzeugt.
- Die Bewegung ist auf zwei Hauptrichtungen zu programmieren:
  links = Verkleinerung der x-Koordinate
  rechts = Vergrößerung
- Die Häufigkeit des Ansprechens auf die Joystickeingabe ist für die Bewegungsgeschwindigkeit von Bedeutung. Bauen wir hier keine "Bremse" ein, dann rast der "Pinsel" schon bei kurzem Joystickausschlag von einer Ecke in die andere.

Man sieht, selbst an so einem einfachen Beispiel gibt es eine ganze Reihe wichtiger Punkte zu beachten. In diesem ersten Fall wollen wir deshalb besonders langsam vorgehen, um nicht gleich Verständnisschwierigkeiten zu provozieren.

Wir zergliedern nun unser Hauptprogramm in mehrere Teile und lassen zunächst einmal die Festlegung der Anfangszustände weg. (Das erfolgt im nächsten Schritt.)

Teil 01: Warteschleife
Für die Warteschleife verwenden wir zwei Adressen (1008/1009), die wir als Zähler in der Reihenfolge LO/HI benützen. Je nachdem wie hoch wir die Grenzwerte setzen, erfolgt ein Einsprung ins Programm. Hat der Zähler seinen vorgegebenen Wert noch nicht erreicht, dann wird er lediglich um 1 erhöht und wieder an den Anfang der Warteschleife verzweigt.
In unserem Beispielprogramm verlangen wir, daß HI=10 und LO=100 ist. Es muß also bis 10 mal 256 + 100 = 2660 hoch gezählt werden, bevor der Sprung zum Weitermachen freigegeben wird.

Teil 02: Joystickabfrage
Das Hauptprogramm beginnt mit der Abfrage des Joysticks. Dabei werden alle Richtungen berücksichtigt, die eine Links- oder Rechtsanzeige zur Folge haben könnten. Wird ein Joystickausschlag erkannt, dann erfolgt ein Sprung zum Programmteil "Links-" oder "Rechtsausschlag".

Teil 03: Rechtsausschlag
Zunächst erfolgt eine Prüfung, ob der rechte Anschlag schon erreicht ist oder nicht.
Liegt noch kein Vollausschlag vor, dann muß sich das Sprite nach rechts bewegen. Wir lassen ihn dabei um 4 Pixel wandern. Das entspricht einer halben Cursorbreite.
Die Vervierfachung bewerkstelligen wir mit dem Assembler-Befehl ASL, den wir zweimal hintereinander anwenden.
Natürlich ist es eleganter, den Pinsel um nur eine Pixelbreite zu verschieben. Aber wir wollen auch mal andere Möglichkeiten ausprobieren.
Immer wird aber nur die x-Koordinate des Sprites verändert, die y-Richtung bleibt konstant.

Teil 04: Linksausschlag entspricht dem Teil 03

Teil 05: Vorläufiger Abschluß
Nach dem Verschieben des Pinsels werden die Zähler in (1008/1009) wieder auf Null gestellt, um nicht sofort wieder eine weitere Verschiebung auszulösen. Anschließend erfolgt der Sprung an den Programmanfang.
Erweiterungen dieses Hauptteils 01 bis 05 können später jederzeit hier angefügt werden. Lediglich der Rücksprung muß ans Ende gestellt werden.

2.1.4 ASSEMBLER-Beispiel für "01-pinsel" (C64):

```
01 18000 ldy 1009     ; HI des Zählers laden und
   18003 cpy #10      ; auf Obergrenze überprüfen
   18005 beq 18017    ; erreicht? ja ===> LO auf Grenze prüfen
   18007 inc 1008     ; nein ---> Zähler LO erhöhen
   18010 bne 18000    ; und falls kein Überlauf ---> zum Anfang
   18012 inc 1009     ; falls Überlauf ---> Zähler HI erhöhen
   18015 bne 18000    ; und unbedingt an den Anfang springen

   18017 inc 1008     ; Zähler LO erhöhen
   18020 ldy 1008     ; Zähler LO laden
   18023 cpy #100     ; und mit Grenzwert vergleichen
   18025 bne 18017    ; nicht erreicht ===> LO weiter erhöhen
         nop
```

```
02 Hauptschleife
   18028 lda 56320    ; PADAT abfragen (Daten des C-Ports)
   18031 bit 180      ; Bit 3 gelöscht?  Linksausschlag?
   18033 beq 18059    ; ja ===> zur Linksroutine
   18035 and #8       ; Prüfung von Bit 4
   18037 bne 18083    ; gesetzt ---> kein Rechtsausschlag ===>
         nop            Sprung zur Eingabeabfrage

03 Ausschlag nach rechts
   18040 lda 189      ; AUSSCHLAG laden
   18042 cmp #32      ; bereits Vollausschlag rechts?
   18044 beq 18083    ; ja ===> neue Eingabe abwarten
   18046 inc 189      ; sonst AUSSCHLAG erhöhen
   18048 lda 189      ; und laden
   18050 asl          ; Wert verdoppeln
   18051 asl          ; ... vervierfachen
   18052 sta 53252    ; und als neue x-Koordinate nach XO2
   18055 clc          ; und Sprung ans Ende 05
   18056 bcc 18075
         nop
04 Ausschlag nach links
   18059 lda 189      ; auf Vollausschlag links
   18061 cmp #4       ; prüfen
   18063 beq 18083    ; erreicht ===> auf nächste Eingabe prüfen
   18065 dec 189      ; AUSSCHLAG um eins zurück
   18067 lda 189      ; und laden
   18069 asl
   18070 asl          ; ... vervierfachen
   18071 sta 53252    ; neue x-Koordinate nach XO2
         nop
05 18075 lda #0       ; Zähler
   18077 sta 1008     ; LO und
   18080 sta 1009     ; HI auf Null stellen
   18083 clc
   18084 bcc 18000    ; und zurück zur Warteschleife
```

Anmerkung:
Auch die Spritenummern beginnen beim Durchzählen mit Null. Sprite 2 ist also das dritte mögliche Sprite, weil das erste Sprite die Nummer 0 trägt.
Das entspricht der Zähl- und Schreibweise bei den Bits. Das achte Bit ist z.B. Bit 7.

Speichern Sie dieses Programm nun unter "01-pinsel" ab, lassen Sie es aber noch nicht laufen, denn wie schon gesagt, sind vor dem Start noch einige Vorbereitungen notwendig, die wir jetzt anpacken wollen.

2.1.5 Vorbereitungen, Initialisierungen

Vor dem Start müssen die Anfangszustände definiert werden. Wir haben dabei folgende Bedingungen zu erfüllen:

- Die Zähler für die Warteschleife müssen auf Null gestellt werden, um ein ordnungsgemäßes Anlaufen zu gewährleisten.
- Der Index **AUSSCHLAG** muß in die Mitte, also auf 18 gestellt werden (Mitte zwischen 4 und 32).
- Als aktives Sprite wählen wir Sprite 2 (also das 3. Sprite), das mit Bit 2 in der Adresse (53269) aktiviert wird.
  Diese Wahl ist rein willkürlich. Jedes andere Sprite ist genau so gut geeignet.
- Die Daten von Sprite 2 legen wir ab Adresse (832) an und belegen daher Adresse (2040+2) mit 13, da in 64er-Schritten nach dem Beginn der Sprite-Bytes gefragt wird. (13 mal 64 = 832).
- Die X-Koordinate dieses Sprites belegen wir mit 98, die Y-Koordinaten mit 72. Damit liegt die Mittelstellung im linken oberen Bildschirmteil.
- Mit einer Schleife belegen wir im Wechsel 48 Bytes für Sprite 2 mit 0,255,255. Den Rest füllen wir mit Nullen auf. Das ergibt einen Sprite, dessen linker und unterer Rand transparent ist und dessen rechter oberer Teil das sichtbare Quadrat darstellt.
- Als Farbe wählen wir mit Nummer 5 ein Dunkelgrün in Adresse (53289).
- Der Abschluß dieses vorläufigen Initialisierungsprogramms bildet der absolute Sprung an den Anfang des Hauptprogramms, also nach (18000).

Bezeichnen wir diesen Programmteil als "02-vorpinsel" und schauen uns das in ASSEMBLER an:

ASSEMBLER-Beispiel zu "02-vorpinsel" (C64):

```
- 16000 lda #0       ; Null
  16002 sta 1008     ; nach Zähler LO
  16004 sta 1009     ; und Zähler HI
```

```
- 16008 lda #18      ; Mittelstellung
  16010 sta 189      ; nach AUSSCHLAG

- 16012 lda #5       ; Farbe dunkelgrün
  16014 sta 53289    ; nach COL02, Farbregister für Sprite 2

- 16017 lda #13      ; Sprite-Pointer für Sprite (mal 64)
  16019 sta 2042     ; nach SP02 für Sprite 2

- 16022 lda #72      ; Y-Koordinate (konstant)
  16024 sta 53252    ; nach X02
  16027 lda #98      ; X-Koordinate (Mittelstellung)
  16029 sta 53253    ; nach Y02

- 16032 ldx #47      ; Zähler für 48 Bytes
  16034 lda #255     ; alle Bits gesetzt
  16036 sta 832,x    ; Pixelreihe setzen
  16039 dex
  16040 sta 832,x    ; nächstes Byte belegen
  16043 lda #0       ; Null, also alle Bits gelöscht
  16045 dex
  16046 sta 832,x    ; Pixelreihe löschen
  16049 dex
  16050 bpl 16034    ; Zähler nicht negativ ===> weitermachen
  16052 ldx #16      ; neuer Zähler für den Rest von Sprite 2
  16054 lda #0       ; Rest
  16056 sta 880,x    ; mit Nullen belegen
  16059 dex
  16060 bpl 16056    ; Zähler nicht negativ ===> weitermachen

- 16062 lda #4       ; 3.Bit für Sprite 2
  16064 sta 53269    ; in SPRAKT setzen
  16067 sta 180      ; Bit 3 in (180) setzen für BIT-Operation
- 16070 jmp 18000    ; und Sprung zum Hauptprogramm "Öl-pinsel"
```

Starten wir nun endlich unser Programm (ab 16000). Es sollte nun ein grünes Quadrat auf dem Schirm erscheinen, das mit dem Joystick nach links und rechts verschoben werden kann.

Passen Sie auf, wenn Sie das Programm von BASIC aus mit SYS 16000 starten: Wenn Sie die STOP-Taste nicht abfragen, kommen Sie nur noch durch ein RESET aus dem Programm heraus.
Hoffentlich haben Sie also alle Programmteile vorher abgespeichert, bevor Sie ans Ausprobieren gingen? (nur 40/80XX)

2.1.6 Verwendete Labels - Zusammenstellung

Label	C64	40/80XX
AUSSCHLAG	189=$bd	189=$bd
COLO2	53289=$d029 Farbregister für Sprite 2	-----
SPO2	2042=$07fa Anfangsadresse der Daten von Sprite 2	-----
XO2 YO2	53252=$d004 53253=$d005 x- bzw. y-Koordinaten von Sprite 2	----- -----
SPRAKT	53269=$d015 Register für aktive Sprites (bitweise)	-----

2.2 Variieren und Testen des "pinsel"-Programms

2.2.1 Variable Laufgeschwindigkeit

Die Geschwindigkeit, mit der Sie den Pinsel über den Bildschirm steuern können, läßt sich in einem breiten Bereich wählen: von blitzartig bis sehr langsam.
Verändern Sie dazu die CPY #...-Befehle in der Warteschleife. Die Grobabstimmung nehmen Sie mit dem HI-Zähler aus (1009) vor, die Feinabstimmung mit dem LO-Zähler aus (1008).
Am schnellsten geht es natürlich, wenn Sie beide schon beim Inhalt 0 zur Hauptroutine schicken.

2.2.2 Bildschirmobjekte austauschen

Sie können nun das Sprite anders gestalten und ihn als Auto, Flugzeug oder Schiff über den Bildschirm jagen.
Wenn Sie auch noch die Y-Koordinate über den Joystick steuern, dann steht Ihnen der ganze Bildschirm offen und noch mehr, weil Sie auch darüber hinaus in einem nicht sichtbaren Bereich weiterfahren können. Lesen Sie darüber in Ihrer C64-Literatur nach.

2.2.3 Bildschirmgestaltung

Bleiben wir noch kurz bei unserem "pinsel"-Programm.
Um die Ausschläge auch gut ablesen zu können, stellen wir noch eine Art Skala dar, die aus drei Quadraten oberhalb des Pinsels besteht. Das mittlere Quadrat gibt die Mittelstellung an, wenn der Pinsel genau darunter steht. Die äußeren Quadrate werden mit einer Pinselbreite Abstand zum mittleren gezeichnet, so daß wir die Richtung nun in der Einheit "Pinselbreite" ablesen können.

Wir erweitern die Initialisierungs-Routine "02-vorpinsel", indem wir vor dem JMP 18000 folgendes einfügen:

```
- 16069 lda #160     ; Code für Cursorzeichen (revers leer)
  16071 sta 1231     ; rechts außen oben
  16074 sta 1232
  16077 sta 1191     ; rechts außen unten
  16080 sta 1192
  16083 sta 1227     ; Mitte außen oben
  16086 sta 1228     ;
  16089 sta 1187     ; links außen oben
  16092 sta 1188
  16095 sta 1235     ; rechts
  16098 sta 1236
  16101 sta 1195
  16104 sta 1196
- 16107 lda #13      ; Farbe hellgrün
  16109 sta 55503    ; Belegung der entsprechenden Farbspeicher
  16112 sta 55504    ; ...
  16115 sta 55463
  16118 sta 55464
  ...
  16142 sta 55468
  16145 jmp 18000    ; Sprung zum Hauptprogramm
```

Sie können diesen Programmteil beliebig erweitern, indem Sie ein Gehäuse für dieses Anzeigeinstrument entwerfen, einen Rahmen zeichnen oder eine Beschriftung entwerfen.
Am Schluß steht aber immer der Sprung nach (18000).

Speichern Sie dieses Programm auf jeden Fall ab unter dem Namen "03-vorpinsel". Wir werden es später noch einmal zusammen mit dem Programm "01-pinsel" verwenden.

2.3 Abweichungen für die 40/80XX-Geräte

Wer mit den großen Geräten arbeitet, muß leider auf die komfortable Sprite-Edition verzichten. Aber auch hier ist der Wendezeiger ohne weiteres darstellbar.
Die Warteschleife 01 bleibt erhalten wie sie vorgestellt wurde.
Beim Abfragen der Eingabe kann man die numerische Tastatur verwenden, indem man die Zeropage-Adresse (151) bzw. (155) beim SK untersucht. Ist keine Taste gedrückt, steht dort 255.

Um den Pinsel zu versetzen, nehmen wir auf der einen Seite eine Cursorbreite weg und setzen sie auf der anderen wieder an, wenn die Randwerte nicht überschritten wurden.
Wir gehen davon aus, daß die Lage des Pinsels durch den Inhalt der Zeropageadresse **AUSSCHLAG** = (189) indiziert ist.
Wenn der Pinsel eine Breite von 4 und eine Höhe von 2 Cursorflecken hat, ergibt das auf dem 80-Zeichenschirm ebenfalls ein Quadrat.
Die Grundadressen für die linken Seiten sind dann zum Beispiel (33651) und (33731), für die rechten Seiten (33654) und (33734).

Teil 03 sieht dann etwa für den 80-Zeichenschirm so aus:

```
03 18054 ldx 189      ; Anschlag bei Grenze rechts
   18056 cpx #39      ; erreicht?
   18058 beq 02
   18060 lda #32      ; Leerzeichen laden
   18060 sta 33651,x  ; auf der linken Seite ausgeben
   18063 sta 33731,x  ; und eine Zeile tiefer
   18066 inx          ; Stellung erhöhen
   18067 lda #160     ; Cursorfleck laden
   18069 sta 33654,x  ; und rechts oben ansetzen
   18072 sta 33734,x  ; und eine Zeile tiefer ebenfalls
   18075 stx 189      ; neuen AUSSCHLAG abspeichern
   18077 bne Teil 05  ; und Sprung zum Abschlußteil 05
```

Die "04-Linksroutine" verläuft entsprechend.

Beim Initialisieren entfallen die Spritevorbereitungen.
Die Rücksetzung der Zähler bleibt wie sie ist. Die Grundstellung des Pinsels läßt sich durch einfaches Ausgeben des Codes 160 an die gewünschten Stellen zeichnen.
Statt einer anderen Farbe kann man beim Aufbau der Skala-Symbole

den Code 102 ausgeben. Dann hebt sich der bewegliche Pinsel gut von der Skala ab.
Zweckmäßigerweise schaltet man den Computer auf Graphikdarstellung um, damit der Abstand zwischen den Zeilen verschwindet und der Pinsel ein zusammenhängendes Bild liefert.
Die Umstellung erfolgt mit JSR 57371. Siehe dazu Kapitel 11.
Der Phantasie sind mit der Blockgraphik im Gegensatz zum C64 enge Grenzen gesetzt. Jedoch ist auch hier ein Rahmen und eine nette Gestaltung möglich.

2.4 Erweiterungs- und Übungsaufgaben:

Probieren Sie nun Ihr Werk aus und versuchen Sie sich an allen möglichen Änderungen:
- Legen Sie den Wendezeiger genau in die Mitte des Schirms.
- Wechseln Sie Form, Größe oder Farben.
- Setzen Sie unter das Pinselquadrat, das ja nur das Ende des Zeigers bedeutet, eine angedeutete Anzeigenadel.
- Verändern Sie die Grenzwerte.
- Lassen Sie beim Aufnehmen der Joystickwerte links und rechts doppelt so viele Einheiten zu wie der Pinsel auf dem Schirm darstellen kann. Sie haben dann zwei Begrenzungen, eine sichtbare und eine unsichtbare.
  Beim Ausschlag nach rechts ist z.B. durchaus der Wert 180 möglich, der auch in **AUSSCHLAG** abgelegt wird. Eine Anzeige erfolgt aber z.B. nur bis 160 bei einer Mittelstellung von 140.

Es lohnt sich, hier ein bißchen zu experimentieren. Das Problem ist überschaubar und für jeden weiteren Ausbau gut geeignet.
Verzweifeln Sie nicht, wenn nicht gleich alles klappt, sondern schalten Sie - nach dem Abspeichern natürlich - Ihr Gerät einfach aus und gehen Sie erst einen oder zwei Tage später wieder an das gleiche Problem. Sie werden erstaunt sein, wie locker Sie plötzlich programmieren können.

3

Erweiterung der Interrupt-Routine – ein Beispiel

3 Erweiterung der Interrupt-Routine - ein Beispiel-

3.1 Die IR-Routine

Die CBM-Rechner unterbrechen 60 mal in der Sekunde alle Programmläufe, um wichtige ROM-Routinen auszuführen, die einen geregelten Computerbetrieb gewährleisten. So wird z.B. die Tastatur abgefragt, die Uhr nachgestellt usw.

Voraussetzung für die Aktivität dieser betriebsinternen Routine ist, daß der sog. IRQ-Vektor auf den Anfang der IR-Routine zeigt. Dies ist ein Zeiger in der Zeropage, der folgende Adressen (LO/HI) hat:

Label	C64	40/80XX
IRQVEC	<788/789>=59953 <$0314/0315>=$ea31 Zeiger mit Inhalt auf den Standard-Interrupt	<144/145>=58453 <$90/91>=$e455
IRQVEC LO	<788>=49 <$0314>=$31 LO des IRQ-Vektors	<144>=85 <$90>=$55
IRQVEC HI	<789>=234 <$0315>=$ea HI des IRQ-Vektors	<145>=228 <$91>=$e4

Erläuterungen dazu:
IRQVEC ist ein Pointer, der auf die Einsprungadresse für den Standard-Interrupt zeigt. Beim C64 z.B. beginnt diese Routine ab (59953). Das ist in LO/HI zerlegt eben 49/234. Der IRQ-Vektor muß also in (788) den Wert 49 und in (789) den Wert 234 enthalten, dann wird automatisch der normale Interrupt durchgeführt.

Verstellt man diesen Zeiger um 3 Adressen nach oben - das wäre beim C64 also (59956) - dann wird die STOP-Tastenabfrage übergangen. Das bedeutet, daß man ein Programm, das mit diesem Interrupt läuft, nicht mehr über die STOP-Taste abbrechen kann.

3.2 Erzeugung eines Taktes mit der IRQ-Abfrage

Zielsetzung:
Wir wollen in einem bestimmten Rhythmus ein Programm durchlaufen. Oder anders ausgedrückt: In einem vorgegebenen Takt sollen bestimmte Abläufe (z.B. Standortbestimmungen, Anzeigen usw.) ausgeführt werden.

Für unser folgendes Beispiel nehmen wir uns vor, daß alle 0,25s ein Flag gesetzt wird, an Hand dessen ein Programm selbst "entscheiden" kann, ob es starten soll oder nicht.
Dadurch lassen sich z.B. Bewegungsabläufe in kleine Schritte zerlegen, so daß man jederzeit die Position o.ä. mitrechnen kann. Es läßt sich somit eine Digitalisierung erzielen, die bei fast allen unregelmäßigen Vorgängen zur Analysierung der Einzelfaktoren erforderlich ist.
Nennen wir das dafür zuständige Programm "04-taktmodul".

3.3 Wichtige Adressen von "04-taktmodul"

- Wir lassen die Interrupt-Erweiterung bei (17000) beginnen.
- Als Flag verwenden wir die Adresse (1010).

Wir vereinbaren, daß wir (1010) mit 1 belegen, wenn eine Zeit von 0,25s verstrichen ist. Läuft auf Grund dieses Flags ein Programm an, muß dieses Programm selbst dafür sorgen, daß (1010) wieder auf 0 zurückgesetzt ist.
(Voraussetzung für die Einhaltung eines gleichmäßigen Takts ist dann, daß das dadurch gestartete Programm auch bei allen möglichen Verzweigungen nie länger als 0,25s für einen Durchlauf benötigt.)

Die Warteschleife lassen wir wieder über einen Zähler laufen, für den wir die Adressen (1011/1012) als LO/HI bereitstellen. Damit haben wir wieder die Möglichkeit, unsere Taktfrequenz in einem relativ großen Bereich vorzuwählen.

3.4 Ablauf von "04-taktmodul"

01: Warteschleife
- Erhöhen des Zählers in (1011/1012).
- Abfrage, ob der gewünschte Wert erreicht ist.

02: Taktflag
- Setzen des Taktflags (1010) auf 1, wenn die vorgegebene Zeit erreicht ist.

03: Abschluß
- Weitersprung zur betriebsinternen IR-Routine.

3.5 ASSSEMBLER-Programm für "04-taktmodul"(C64):

01:Warteschleife:

```
17000 inc 1011       ; Zähler LO erhöhen
17003 bne 17008      ; ungleich 0 ===> weiter
17005 inc 1012       ; sonst Zähler HI erhöhen
17008 lda 1012       ; Zähler HI...
17011 cmp #0         ; ...mit 0 vergleichen (Beispiel!)
17013 bne 17037      ; ungleich 0 ===> weiter
17015 lda 1011       ; Zähler LO...
17018 cmp #15        ; mit 15 vergleichen (Beispiel für 0,25s)
17020 bne 17037      ; nicht erreicht ===> weiter
17022 nop
```

02: Taktflag:

```
17023 lda #1         ; Flag
17025 sta 1010       ; in (1010) auf 1 setzen
17028 lda #0         ; Zähler
17030 sta 1011       ; ...LO
17033 sta 1012       ; ...und HI zurücksetzen
```

Teil 3:

```
17036 jmp 59953      ; Sprung zur CBM-IR-Routine
```

Wir erhalten einen 0,25s-Takt, wenn wir mit dieser Schleife warten, bis der Zähler in (1011/1012) auf 0/15 steht.
Das läßt sich durch das angekündigte Testprogramm überprüfen.

Ein genaueres Intervall für exakte Rechenroutinen können wir auch mit Hilfe der Stoppuhr ermitteln. Im Normalfall genügt aber eine Genauigkeit von 1/60 Sekunde.

3.6 Abstimmung des Taktes

Die Warteschleife (sie entspricht der des vorhergehenden Moduls) muß immer individuell abgestimmt werden. Das geschieht mit Hilfe des Zählers in (1011/1012).
Wann eine Zeit von 0,25s erreicht ist, wird durch ein kleines Testprogramm ermittelt.

Für diesen Minitest lassen wir uns immer dann, wenn das Intervall abgeschlossen ist, ein beliebiges Zeichen auf dem Bildschirm ausgeben und setzen danach sofort unseren Zähler und das Flag auf 0.

3.7 Eine kleine Testroutine für die Bestimmung der Taktfrequenz

Das folgende Programm (ab 17100) fragt zunächst eine Taste ab, mit deren Hilfe wir den Testlauf abbrechen können. Danach wird zu Beginn eines jeden Taktes der Inhalt der Adresse (165) auf dem Bildschirm ausgegeben. Die Bildschirmadresse wird jeweils durch das (X)-Register indiziert und vor jeder neuen Ausgabe inkrementiert (um eins erhöht), so daß wir also maximal 256 hintereinander liegende gleiche Zeichen zu sehen bekommen.

Dann fängt alles wieder an derselben Anfangsposition an. Damit sich das vom Vorhergehenden unterscheidet, wechseln wir einfach den Bildschirmcode, indem wir ihn ebenfalls um 1 erhöhen, wenn die Schleife wieder von vorn beginnt. Auf diese Weise läßt sich durch Mit- oder Abzählen der ausgedruckten Zeichen eine durchschnittliche Taktfrequenz mit guter Genauigkeit bestimmen.

Nach jeder Bildschirmausgabe muß das Taktflag zurückgesetzt werden. Danach erfolgt wieder der Sprung an die Warteschleife für das Taktflag.

Das Takt-Testprogramm "test/takt" (C64)

```
- 17100 lda 203
  17102 cmp #60         ; Stoptaste (Leertaste) gedrückt?
  17104 bne 17108       ; nein ===> weiter
  17106 rts             ; ja ===> Rücksprung
  17107 nop
- 17108 lda 1010
  17111 beq 17100       ; Flag 0 ===> warten
  17113 inx             ; Flag gesetzt ---> Zähler erhöhen
  17114 bne 17118       ; 255 überschritten? nein ===> weiter
  17116 inc 165         ; sonst Zeichencode erhöhen
  17118 lda 165         ; Zeichencode laden
  17120 sta 1024,x      ; und auf Bildschirm ausgeben
- 17123 lda #0
  17125 sta 1010        ; Flag auf 0
  17128 beq 17100       ; und Sprung an den Anfang der Schleife
```

Wenn sie das Programm "04-taktmodul" noch im Speicher haben, sollten Sie es jetzt zusammen mit diesem "test/takt" abspeichern. Nennen wir es einfach "05-takt".
Es läßt sich auf diese Weise später bequem - auch von BASIC aus - mit einem einzigen LOAD-Befehl einladen.

Lauffähig ist es allerdings noch nicht, da wir den Taktgeber, der ja im Programm "04-taktmodul" steckt, noch nicht aktiviert haben. Er muß jetzt erst in die Interrupt-Routine integriert werden. Es bringt also im Moment noch gar nichts, wenn Sie bei (17000) oder (17100) starten. Gleich im nächsten Abschnitt packen wir dieses Problem an.

3.8 Einstellen des IRQ-Vektors

Die Interrupt-Routine muß nun unser "04-taktmodul" durchlaufen. Das läßt sich dadurch erreichen, daß wir den vorhin besprochenen IRQ-Zeiger auf die Anfangsadresse dieses Programms einstellen. 17000 hat zerlegt in LO/HI die Werte 104/66. Setzen wir die nun in die Adressen des IRQ-Zeigers **IRQ-LO** und **IRQ-HI**, dann wird die Interrupt-Routine brav bei (17000) beginnen.

Allerdings hat die ganze Sachen noch einen kleinen Haken, der

sich aber schnell geradeklopfen läßt:
Der IRQ-Zeiger läßt sich nicht durch einfaches "poken" der Zero-page-Adressen ändern, weil ja während dieser Durchführung auch der Interrupt ausgeführt wird. Er kann deshalb auch zwischen das Einstellen von LO und HI des Zeigers fallen und schon stürzt der Rechner ins Leere, weil er einen Interrupt-Einsprung durchführt, aus dem er in der Regel nicht mehr herauskommt.

Wer den Befehl DOKE besitzt, umgeht diese Schwierigkeit und kann den IRQ-Zeiger z.B. mit DOKE144,17000 auf (17000) einstellen. Ansonsten muß vor einer Veränderung des Vektors, die sowohl das LO- als das HI-Byte betrifft, das "Interrupt-Disable-Flag" mit SEI gesetzt werden. Der Interrupt bleibt also aus, der Zeiger kann in Ruhe verändert werden. Anschließend setzt man dieses Flag wieder mit CLI zurück und ermöglicht damit den Einsprung in die Adresse, auf die der Vektor **IRQVEC** zeigt.

Das folgende kleine Programm - der Wichtigkeit halber auch gleich für die "großen" Geräte gelistet - stellt den IRQ-Vektor ein.

ASSEMBLER-Beispiel "06-irqvec17000"

C64	40/80XX	
- 24500 sei	sei	; I-Flag setzen
- 24501 lda #104	lda #104	; LO
24503 sta 788	sta 144	; nach IRQVEC LO
24505 lda #66	lda #66	; HI
24507 sta 789	sta 145	; nach IRQVEC HI
- 24509 cli	cli	; I-Flag löschen
- 24510 jmp 17100	jmp 17100	; zu "test-takt"

Jetzt können Sie das Programm gleich mit SYS 24500 starten. Sie müssen aber das Programm "05-takt" noch im Speicher belassen haben.

Zunächst wird der IRQ-Vektor verstellt und dann zum Programm "test-takt" verzweigt.
Ab sofort wird ca.60 mal pro Sekunde das Programm "04-taktmodul" (17000) angesprungen, und der Zähler wird solange inkrementiert, bis er die eingestellte Zahl erreicht hat.

Unabhängig davon läuft das Testprogramm "test-takt" ab 17100 solange auf der Stelle, bis das Taktflag gesetzt ist. Erst wenn hier eine 1 vorliegt, erfolgt eine Zeichenausgabe auf dem Schirm.

Nun können Sie die Anzahl der gesetzten Zeichen und die dazu benötigte Zeit feststellen und Sie erhalten Ihre Taktfrequenz.

Verändern Sie mit POKE 17019,XX einmal Ihren Zähler, dann werden Sie bald merken, daß Sie mit XX=15 recht gut einen 0,25s-Takt erhalten.

Anmerkung:
Daß wir die Taktfrequenz von BASIC aus überprüfen, hat seinen Grund darin, daß einige Assembler ebenfalls die Adressen des IRQ-Zeigers verwenden, so daß eine Veränderung nicht ratsam ist.

Zusammenfassung:

Die Interrupt- Erweiterung ist nun so aufgebaut, daß Sie den Zähler beliebig einstellen können zwischen ca. 1/60 Sekunde und ca. 18 Minuten.

Jeder weitere Ausbau dieser Routine führt allerdings dazu, daß etwas mehr Zeit zur Ausführung benötigt wird, aber in Maschinensprache haben wir noch einige Reserven.

Die ganze Geschichte mit dem Taktgeber hat auch ihren Bezug zum vorhergehenden Kapitel, wo wir einen Wendezeiger simuliert haben. Wir sind nämlich jetzt (fast) in der Lage, mit Hilfe dieses Geräts die Richtung anzugeben, in die wir uns "bewegen".
Wenn wir den Ausschlag des "Pinsels" aus dem Programm "Ol-pinsel" in regelmäßigen kurzen Abständen untersuchen, dann läßt sich daraus eine Richtungsänderung berechnen.
Voraussetzung dazu ist aber, daß wir sehr schnell rechnen können. Und das wiederum erfordert den Einsatz von Maschinenprogrammen. In den nächsten beiden Kapiteln Nr.4 und Nr.5 werden wir uns ausführlich mit der Arithmetik befassen. Sie ist einer der Schwerpunkte in dieser Schrift.

Doch vorher sollten Sie sich zunächst einmal an einer der folgenden Übungen versuchen.
Zwischendurch muß immer wieder betont werden, daß das bloße Durchlesen von bereits fertigen oder schon besprochenen Programmen nicht allzu viel Lerneffekt mit sich bringt. Mehr als in anderen Bereichen gilt hier das alte Prinzip: Probieren - wobei damit immer systematisches Probieren, nicht aber Herumprobieren gemeint ist - geht über (oder zumindest neben) Studieren.

Aufgaben:
Erweitern Sie die Interrupt-Routine, um die Tastatur abzufragen und z.B. beim Drücken der Taste "0" sofort ein Flag zu setzen, das aber nach 5 Sekunden wieder gelöscht wird.

Erweitern Sie die Interrupt-Routine, um die Eingaben des Joysticks abzufragen und in der richtigen Reihenfolge in einem bestimmten Adressbereich bereitzustellen (Pufferproblem!).

4

Zahlen, Variablen – Formate

4 Zahlen, Variablen - Formate

4.1 Rechnen mit Ganzzahlen (Integer-Zahlen)

Die CBM-Rechner verarbeiten Integerzahlen in Form von Zweibyte-zahlen, die in der Reihenfolge HI/LO den Zahlenwert darstellen. Steht in der kleineren Adresse z.B. 10 und in der nächsthöheren z.B. 20, dann ergibt das einen Zahlenwert von 10 mal 256 + 20 = 2580.

Der Bereich ist dadurch auf insgesamt 65535 Zahlen beschränkt und zerfällt in einen negativen Bereich (bis -32768) und einen positiven (bis 32767).
Bei CBM-internen Umwandlungen und Ausgaben wird eine negative Zahl daran erkannt, daß im HI-Byte das Bit 7 gesetzt ist, das Byte also einen Wert von mindestens 128 hat.

Solange man nicht die CBM-eigenen Routinen für die Ausgabe eines Integererqebnisses verwendet, lassen sich Addition und Subtraktion im Zwei-Byte-Verfahren von 0 bis 65535 problemlos durchführen.
Anders ausgedrückt: Will man ROM-Routinen zur Ausgabe verwenden, dürfen Zwischen- oder Endergebnisse den Bereich von -32767 bis +32767 nicht verlassen.

Addition und Subtraktion können in ASSEMBLER mit den Befehlen CLC/ADC bzw. SEC/SBC im genannten Bereich durchgeführt werden.

4.1.1 Rechnen mit positiven Ganzzahlen

Nehmen wir an, die Zahl(%) 500 steht in (17000/17001) und wir wollen 300 addieren, was in (17002/17003) abgelegt ist.

Teil 1: Erzeugen der Integerzahlen
Dann belegen wir die Adressen (17000) bis (17003) wie folgt:

```
<17000>=1   ; HI von 500
<17001>=244 ; LO von 500
<17002>=1   ; HI von 300
<17003>= 44 ; LO von 300
```

Soll das Ergebnis der Addition 500+300 wieder in (17000/17001) stehen, dann sieht der nächste Programmschritt so aus:

Teil 2:
Wir addieren zunächst die beiden LO-Bytes und erhalten 244+44=288. Dies ist jedoch in Byteform nicht mehr darstellbar, weil nur bis 255 hochgezählt werden kann.
Der Prozessor zählt zwar brav weiter, fängt dann aber nach 255 wieder mit 0 an, so daß im (A)-Register nun 32 steht. Gleichzeitig wird aber das C-Flag gesetzt, an dem man den "Überlauf" erkennen kann.
(Daher muß vorher mit CLC das C-Flag gelöscht werden.)

Teil 3:
Mit Hilfe der Flags läßt sich nun feststellen, ob ein Überlauf stattgefunden hat oder nicht. Die Additions- bzw. Subtraktionsbefehle ADC und SBC beeinflussen:

das N-Flag (negatives Ergebnis ---> <N>=1)
das Z-Flag (Ergebnis = 0 ---> <Z>=1)
das C-Flag (Überlauf ---> <C>=1)
das V-Flag (Bit 7 gesetzt ---> <V>=1)

Ist das C-Flag gesetzt, was mit den Branch-Befehlen BCS und BCC untersucht wird, dann erhöhen wir die Adresse, in der das HI-Byte des zweiten Summanden steht oder - falls wir diesen noch in unverändertem Wert benötigen - laden ihn in ein Register und erhöhen dieses, bevor wir die HI-Bytes addieren.

Das Ergebnis der Addition wird irgendwo im RAM abgelegt. In unserem Beispiel verwenden wir dazu gleich die Adressen des ersten Summanden (17000/17001). Allerdings müssen wir uns notieren, wo die Summe zu finden ist, wenn wir sie - z.B. zur Ausgabe auf dem Bildschirm - wieder verwenden wollen.

Man kann dieses Mitverwalten umgehen, wenn man mit Integervariablen arbeitet, die entsprechend dem BASIC-System mit zwei Zeichen benannt sind. Allerdings kommt man dann nicht mehr mit zwei Adressen pro Integerzahl aus, sondern braucht deren sieben. Doch davon lesen wir später mehr.

Zurück zu unserem Beispiel: Zur Ausgabe auf dem Bildschirm verwenden wir die ROM-Routine **INTOUT**, die im Kapitel "Ausgabe-Routinen" noch näher besprochen wird.

Vorab nur so viel: Um **INTOUT** zu verwenden, muß das LO-Byte unseres Ergebnisses im (X)-Register, das HI-Byte im (A)-Register stehen. Da wir wissen, wo sich unser Ergebnis befindet - nämlich in den Adressen (17000) und (17001) ist das kein Problem.

ASSEMBLER-Beispiel:"07-integadd" (C64):

Teil 1: Bereitstellen der Zahlen 500 und 300

```
- 17500 lda #1
  17502 sta 17000       ; <17000/17001>=500
  17505 sta 17002
  17508 lda #244        ; und
  17510 sta 17001
  17513 lda #44         ; <17002/17003>=300
  17515 sta 17003
  17518 nop
```

Teil 2: LO-Bytes addieren

```
- 17519 lda 17001       ; 244 in (A)
  17522 clc
  17523 adc 17003       ; 44 zum (A) addieren
  17526 bcc 17531       ; Carry-Flag gesetzt?  nein ===> Sprung
  17528 inc 17000       ; ja===> HI-Byte von 500 erhöhen
- 17531 sta 17001       ; Ergebnis LO (hier 32) nach (17001)
  17534 tax             ; und für die Ausgabe nach (X)
  17535 nop
```

Teil 3: HI-Bytes addieren

```
- 17536 lda 17000       ; erhöhtes HI-Byte nach (A)
  17539 clc

  17540 adc 17002       ; HI-Byte von 300 zum (A) addieren
- 17543 sta 17000       ; ... und ablegen. (Nicht notwendig, wenn
  17546 nop             ; die Zahl 800 nicht mehr gebraucht wird.
```

Teil 4: Ausgabe der Integerzahl

In unserem Fall ist das LO der Summe bereits mit TAX ins (X)-Register übertragen worden. Das HI steht noch in (A). Ist dies nicht der Fall, dann werden die entsprechenden Bytes erst geholt, bevor sie mit **INTOUT** ausgegeben werden können:

```
- 17547 ldx 17001       ; LO-Byte nach (X)
- 17550 lda 17000       ; HI-Byte nach (A)
- 17553 jsr 48589       ; ROM-Routine INTOUT gibt ab aktuellem
```

```
                          Cursor die Integerzahl (X/A) aus
  17556 rts             ; Rücksprung - Abschluß dieser Routine
```

Achtung:
INTOUT gibt nur positive Zahlen im Bereich von 0 bis 65535 aus.

Speichern Sie das Programm unter "07-integadd" ab und starten Sie es mal von BASIC oder mit dem Assembler bei (17500), bei (17519), bei (17536) und bei (17547).
Haben Sie die ausgedruckten Zahlen auch erwartet?
Wenn nicht, sollten Sie diesen Abschnitt noch einmal nachlesen.

4.1.2 Negative Ganzzahlen

Negative Ganzzahlen erkennt man daran, daß das Bit 7 im HI-Byte der Zahl gesetzt ist. Allerdings läßt sich das Vorzeichen einer Integerzahl nicht ohne weiteres durch Setzen oder Löschen des 7. Bits verändern.
Der Grund liegt darin, daß die negativen Ganzzahlen eben *von 0* abwärts gezählt werden, so daß die Zahl (-1) eben als $FFFF auftritt. Der größte Betrag einer negativen Zahl liegt dann bei $8000, also -32768.
Durch Zerlegen in LO/HI läßt sich der Wert aus der Hexdarstellung ermitteln.

Beispiel:
Gesucht ist der Wert der Zahl $fea0.

Lösung:
HI-Byte ist $fe, 2 Einheiten unter $00, also 2 mal -256 = -512.
LO-Byte ist $a0, also 160.
Ergebnis: $fea0=-512+160=-352

Das LO-Byte ist dabei immer positiv zu behandeln. Das HI-Byte ist nur dann negativ, wenn es gößer oder gleich $80=128 ist. Daher ist $7fff auch die größte positive Integerzahl, denn wenn man nun nochmals um 1 erhöht, erhält man $8000, also eine negative Zahl.

Daraus ergibt sich eine Möglichkeit, das Vorzeichen zu wechseln: Wir invertieren sowohl HI- als auch LOW-Byte der Integerzahl. Dazu eignet sich der ASSEMBLER-Befehl **EOR #255**, der bitweise die Verknüpfung mit dem im (A)-Register stehenden Byte besorgt.

Das sieht dann in Bitschreibweise z.B. für das Byte $FA=250 so aus:

```
          <A> =       11111010     = 250(10)
     EOR #255         11111111
Ergebnis: <A> =       00000101     , also 5 in (A)
```

Zur Erinnerung: EOR erzeugt nur bei ungleichen Bits eine 1.
Es wird also mit EOR immer der Komplementärwert auf 255 erzeugt:
In unserem Beispiel: 250+5=255

Soweit der Ausflug in die Bitverknüpfung EOR, nun zurück zu den Integerzahlen.
Angenommen, wir haben das Vorzeichen der Zahl $fea0 in (17000/17001) zu wechseln, dann gehen wir folgendermaßen vor:

ASSEMBLER-Beispiel "08-intvorz" für C64:

```
01 18023 lda 17000   ; HI
   18026 eor #255    ; invertieren
   18028 tax         ; und vorläufig nach (X) retten
         nop
02 18030 lda 17001   ; LO
   18033 eor #255    ; invertieren,
   18035 tay         ; nach (Y) übertragen und
   18036 iny         ; um 1 erhöhen, sonst Ergebnis falsch um 1

03 18037 bne 18040   ; neues LO=0 (Sonderfall)? nein ===> weiter
   18039 inx         ; ja ---> HI ebenfalls um 1 erhöhen

04 18040 txa         ; endgültiges HI nach (A)
   18041 jsr 45969   ; INTFLP wandelt <Y/A> in eine FLoat-
                       Point-Zahl um
05 18044 jsr 43708   ; FLPOUT druckt: 352
         rts
```

Sie sehen, wir haben eine neue ROM-Routine **INTFLP** aufgenommen. Und das hat folgenden Grund:

Wie schon angedeutet, funktioniert die Ausgabe einer negativen Ganzzahl nicht mehr mit INTOUT.

Wir könnten uns jetzt einen eigenen Algorithmus ausdenken (Feststellung des Vorzeichens, Ausgabe des Vorzeichens, Ausgabe der Zahl mit INTOUT o.ä.), belassen es aber bei der Möglichkeit, die

das Betriebssystem schon bietet:

- Zunächst wird die Integerzahl in eine sog. Fließkommazahl umgewandelt, die das Vorzeichen in jedem Fall enthält.
  Dazu übergeben wir das LO-Byte der Zahl an (Y) und das HI-byte an (A) und springen bei **INTFLP** ein.

- Diese FLP-Zahl kann durch die Routine **FLPOUT** ausgegeben werden.

Probieren Sie nun die eben erarbeitete Routine aus. Vergessen Sie aber nicht: Der zulässige Bereich von -32768 bis 32767 darf nicht verlassen werden!

Die scheinbare Umständlichkeit, mit der die negativen Zahlen verwaltet werden, hat den Vorteil, daß wir die bereits in 4.1.1 geschriebene Integeraddition nun auch auf negative Zahlen ausdehnen können.
Einzige Bedingung ist aber: Die Ausgabe darf nicht mehr mit **INTOUT** erfolgen, sondern mit **INTFLP** und **FLPOUT.**

4.1.3 Subtrahieren von Integerzahlen im positiven Bereich

Die Programmierung einer Subtraktionsaufgabe verläuft entsprechend der Addition, solange keine negativen Zahlen auftreten. Auch hier kann man dann den Bereich von 0 bis 65535 ausnutzen. Zu beachten ist lediglich, daß das C-Flag vor dem eigentlichen Subtraktionsbefehl SBC gesetzt werden muß, wenn man feststellen will, ob man sich auch wirklich 'eins geborgt hat' oder nicht.

Wandeln wir das Additionsprogramm entsprechend um in das Subtraktionsprogramm "09-intsub". Der erste Teil ist identisch mit dem aus "08-integadd", den zweiten und dritten wandeln wir ab:

ASSEMBLER-Beispiel "09-integsub" (C64):

```
...
Teil 2: LO-Bytes verknüpfen:
- 17519 lda 17001
  17522 sec             ; C-Flag setzen
  17523 sbc 17003       ; LO-Byte von (A) subtrahieren
  17526 bcs 17531       ; C-Flag noch gesetzt? ===> weiter
```

```
- 17528 dec 17002       ; nein ===> HI-Byte des Minuenden -1
  17531 sta 17005       ; LO-Byte ablegen in neuer Adresse
- 17534 tax             ; Ausgabe vorbereiten
        nop
```

Teil 3: HI-Bytes verknüpfen:

```
- 17536 lda 17000
  17539 sec
  17540 sbc 17002
- 17543 sta 17004       ; HI-Byte in neue Adresse
```

Teil 4: Ausgabe auf dem Bildschirm wie "07-integadd"

Haben Sie erkannt, was diesmal anders ist?
Richtig, die beiden ursprünglichen Zahlen (Minuend und Subtrahend) bleiben erhalten. Der Differenzwert steht in (17004/17005).

4.1.4 Subtraktion mit beliebigen Integerzahlen

Begnügen wir uns mit dem eingeschränkten Bereich -32768 bis +32767, dann können wir die Subtraktion wie die Addition behandeln.
Denn mathematisch gesehen gilt: a-b = a+(-b)

Daraus leitet sich das anzuwendende Verfahren ab:
- Das Vorzeichen des Subtrahenden wird verändert.
- Nun kann eine Addition durchgeführt werden.

Aufgabe:
Stellen Sie die bisher besprochenen Programmteile zu einem Modul zusammen, mit dem man Integerzahlen addieren und subtrahieren kann.

4.1.5 Höhere Rechenarten mit Integerzahlen

Multiplikation, Division, Potenzieren, Radizieren usw. lassen sich über der entsprechenden Grundmenge ebenfalls mit Integerrechnungen durchführen.
Da man aber z.B. schon bei der Division auf Nicht-Ganzzahlen stößt (wie z.B. bei 5/2), ist dies meist wenig sinnvoll.
Man verwendet in diesen Fällen die reellen Zahlen.

Versuchen Sie aber trotzdem einmal, ein Integer-Multiplikationsprogramm für ein Produkt aus zwei Faktoren zu entwerfen, das nach folgendem Schema arbeitet:

n·z = z+z+z+... (n Summanden z)

Denken Sie daran, daß die Multiplikation nichts weiter ist als eine vereinfachte Addition gleicher Summanden:
z.B. 5·4 = 4+4+4+4+4

Verdoppeln und Halbieren

Um den Inhalt eines Bytes zu verdoppeln, wird es mit dem Befehl ASL behandelt. Dabei werden alle Bits um eine Stelle nach links verschoben.
Aus der Zahl 6_{10}=00000110$_2$ wird dann 00001100$_2$=12_{10}

Würde der Inhalt 255 überschreiten, wird das C-Flag wieder gesetzt und man kann das HI-Byte einer Integerzahl entsprechend erhöhen.

Zahlenbeispiel:
356 ist gegeben mit <17000>=1 und <17001>=100
Gesucht: 2 mal 356 = ?
Das Ergebnis ist wieder in (17000/17001) erwünscht.

ASSEMBLER-Beispiel "10-integdopp" (C64):

```
- 17600 asl 17000     ; HI verdoppeln
  17603 clc
- 17604 asl 17001     ; LO verdoppeln
  17607 bcc 17612     ; kein Übertrag ===> weiter
  17609 inc 17000     ; Übertrag ===> HI erhöhen
  17612 lda 17000     ; HI nach (A) und...
  17615 ldx 17001     ; ... LO nach (X) für die Ausgabe
- 17618 jsr 48589     ; mit INTOUT
  17621 rts           ; Ende
```

Wird diese Routine mehrfach hintereinander aufgerufen, so läßt die in (17000/17001) bereitgestellte Zahl immer wieder verdoppeln, bis der gewünschte Wert erreicht ist.

Nehmen wir an, wir möchten 'mal 16' rechnen, also 4-mal verdoppeln, dann benötigen wir einen Zähler, der nach jedem Verdoppeln

um eins erhöht wird. Verwenden wir dazu eine freie Zeropage-Adresse: (180) ist dafür geeignet.
Man kann auf diese Weise also recht einfach mit den 2er-Potenzen multiplizieren:

1mal asl ---> mal 2
2mal asl ---> mal 4
3mal asl ---> mal 8
und so weiter. Aber bitte - die Grenzen beachten!

ASSEMBLER-Beispiel "ll-integpot":

```
01 17522 lda #0      ; Zähler in (180) auf Null stellen
   17524 sta 180     ; (180) wird als Zähler benützt
02 17526 lda 180     ; Beginn der Schleife: Zähler prüfen
   17528 cmp #4      ; 4. Durchlauf schon durchgeführt?
   17530 beq 17540   ; ja ===> Ende
03 17532 inc 180     ; nein ===> Zähler erhöhen
   17534 jsr 17600   ; ... und 'Verdoppelungs-Routine' aufrufen
   17537 clc
   17538 bcc 17526   ; unbedingter Sprung zum Schleifenanfang 02
   17540 rts
```

Wir haben hier vorausgesetzt, daß die Verdoppelungs-Routine aus dem letzten Abschnitt noch im RAM steht (ab 17600).
Probieren Sie dieses kleine Programm aus, indem Sie es bei 17522 aufrufen (SYS 17522 bzw. EX 17522).
Sie erhalten nach jeder Verdoppelung einen Ausdruck.

Den Bereich von -32768 bis +32768 dürfen Sie allerdings nicht überschreiten. Falls man darauf nicht verzichten kann, muß man eine Drei- oder Vier-Byte-Rechenroutine konstruieren. Die CBM-Ausgabe-Routine funktioniert dann allerdings nicht mehr.

Das Halbieren einer Zahl wird im Prinzip ebenfalls so aufgebaut wie das Verdoppeln. Der entsprechende Befehl LSR arbeitet analog zu ASL, jedoch in die andere Richtung (logic shift right).

Aufgabe 5:
Schreiben Sie ein ASSEMBLER-Programm zur mehrfachen Halbierung einer Integerzahl.

4.1.6 Integermultiplikation mit INTMUL

Die CBM-Routinen enthalten (zur Berechnung der Position eines Feldelements) eine Routine zur Multiplikation zweier Integerzahlen.

Dazu werden die beiden Faktoren F1 und F2 in LO- und HI-Bytes zerlegt und in folgenden Registern bereitgestellt:

Faktor 1 LO/HI in (113/114) beim C64 (bzw (110/111) bei 40/80XX)
Faktor 2 in zwei beliebigen Adressen im Integerformat (HI/LO), deren Anfangsadresse in (95/96) bzw. 80XX:(92/93) erwartet wird.

Faktor F2 kann also eine bereits vorhandene Integerzahl sein. Außerdem muß das (Y)-Register mit 1 belegt werden.

Nach Aufruf der Rechenroutine **INTMUL** steht das Ergebnis im den Registern (X/A) mit (LO/HI) und kann z.B. sofort mit **INTOUT** ausgegeben werden.

Beispiel:
416 mal 60 = ?
Faktor F1: <LO>=150, <HI>=1
Faktor F2: <LO>= 60, <HI>=0
Faktor F2 soll in (20000/20001) stehen, also ab 32(LO)+78(HI)
(zur Erinnerung: 32+(78mal256)=20000)

ASSEMBLER-Beispielt "12-intmul" (C64):

```
01 10000 lda #150   ; LO von F1
   10002 sta 113    ; nach F1-LO
   10004 lda #1     ; HI von F1 nach
   10006 sta 114    ; F1-HI
02 10008 lda #0     ; HI von F2
   10010 sta 20000  ; nach (20000)
   10013 lda #60    ; LO von F2
   10015 sta 20001  ; nach (20001)
03 10018 lda #78    ; HI der Anfangsadresse (20000)
   10020 sta 96     ; nach F2AD-HI
   10022 lda #32    ; LO der Anfangsadresse (20000)
   10024 sta 95     ; nach F2AD-LO
04 10026 ldy #1     ; (Y) als Zähler mit 1 belegen
   10028 jsr 45900  ; INTMUL multipliziert 416 mit 60
```

```
05 10031 jsr 48589  ; INTOUT druckt das Ergebnis: 24360
   10034 rts
```

Wird der Definitionsbereich für Integerzahlen überschritten, dann erfolgt die Fehlermeldung "illegal quantity error", der Rechner geht danach in den READY-Modus.

Zusammenfassung:
Das Rechnen mit Integerzahlen erfordert schon etwas Programmiergeschick. Es wird dort eingesetzt, wo sehr schnell gerechnet werden muß. Der Nachteil: Die CBM-Rechner unterstützen die Integeroperationen kaum. Fast alle Routinen müssen selbst entwickelt werden. Für die allermeisten Fälle sind aber derartige Höchstgeschwindigkeiten nicht notwendig, so daß fast immer die etwas langsameren Verknüpfungen mit reellen Zahlen völlig ausreichen.

Label	C64	40/80XX
F1	113/114=$71/72	110/111=$6f/70
F2AD	(95/96)=($5f/60)	(92/93)=($5c/5d)
INTMUL	45900=$b34c	50295=$$c477

4.2 Arbeiten mit reellen Zahlen

4.2.1 Formate für reelle Zahlen

Reelle Zahlen werden in den CBM-Rechnern in zwei Formen behandelt: dem **Speicherformat** und dem **Registerformat.**

Im Speicherformat wird jede relle Zahl mit 5 hintereinander liegenden Bytes abgelegt, wobei das erste Byte immer den Exponenten darstellt; die nächsten vier Bytes bilden die Mantisse.
Das **Registerformat** unterscheidet sich vom Speicherformat dadurch, daß in einem weiteren 6.Byte das Vorzeichenbit (Bit 7) gesetzt wird.
Registerformatierung wird notwendig, wenn in der Zeropage die reellen Zahlen für Rechenoperationen bereitgestellt werden.
Dafür sind die sog. **Floatpointakkus (FAC)** vorgesehen:

Label	C64	40/80XX
FAC1 :	97... 101=$61...66	97...101=$61...65
FAC2 :	105...110=$69...6e	102...107=$66...6b

Zur Variablenverwaltung wird das **Speicherformat** verwendet, wobei in zwei vorangestellten Bytes der Variablenname geführt wird. Insgesamt benötigt jede reelle Variable also 7 Adressen.
Führen wir die Verwaltung unserer Zahlen bei ASSEMBLER-Programmen selbst, so können wir uns auf die 5 Speicherplätze der reinen Zahl beschränken.

Beispiel (Mehr dazu in Abschnitt 8.1.3):
Die Zahl (-8) steht unter dem Variablennamen "ab" bei (10000).

Speicherformat der Variablen "ab" mit ab=-8:

(10000)	(10001)	(10002)	(10003)	(10004)	(10005)	(10006)
65	66	132	128	0	0	0
"a"	"b"	4+128 (neg.)				

Im Registerformat sieht das dann so aus (z.B. FAC1):

(97)	(98)	(99)	(100)	(101)	(102)
132	128	0	0	0	128

Der Exponent (immer zur Basis 2) wird immer um 128 erhöht abgelegt. Das Vorzeichen "-" erkennt man in Byte 2, wenn hier ein Wert steht, der größer ist als 128 (Bit 7 gesetzt).

Jeder Rechner besitzt in den ROMs eine ganze Reihe von festen reellen Zahlen, die häufig benötigt werden.
Hier die Anfangsadressen für einige wichtige Konstanten:

Label	C64	40/80XX
1,00	47548=$b9bc	51954=$caf2
0,5	48913=$bf11	53477=$d0c7
0,25	58090=$e2ea	54024=$d308
pi	44712=$aea8	48800=$bea0
pi/2	58080=$e2e0	54014=$d2fe
2pi	58121=$e309	54055=$d327
sqr(2)	47579=$b9db	51985=$cb11
-0.5	47584=$b9e0	51990=$cb16
10	47865=$baf9	52271=$cc2f

Schauen Sie mal mit einem Monitor in diese Bereiche hinein, wenn Sie wissen wollen, wie diese Realzahlen aufgebaut sind, ansonsten braucht uns das selten zu interessieren.
Blättern Sie auch mal Ihr ROM-Listing durch, dann finden Sie weitere Konstanten.

4.2.2 Übernehmen von gespeicherten Zahlen mit MEMFAC

Zahlen, die entweder im ROM oder im RAM bereits vorhanden sind, lassen sich auf einfache Weise in den FAC1 übernehmen. Dazu benötigen wir die Anfangsadresse der abgelegten Zahl in der Form LO/HI und übertragen sie nach (A/Y). Anschließend erfolgt der Aufruf von **MEMFAC**.

Beispiel:
Die Zahl 0.25 aus dem ROM (58090 bzw. 54024) soll in den FAC1 übertragen und auf dem Bildschirm ausgegeben werden.

Ablauf und ASSEMBLER-Beispiel "13-memfac" (C64):

```
01 lda #234      ; LO Anfangsadresse
   ldy #226      ; HI Anfangsadresse von 0.25
02 jsr 48034     ; MEMFAC bringt 0.25 nach FAC1
03 jsr 43708     ; FLPOUT druckt:  0.25
   rts
```

4.2.3 Erzeugen von reellen Zahlen

Jede Zahl, die nicht bereits irgendwo in den ROMs vorhanden ist, muß erst einmal im Speicherformat dargestellt und abgelegt werden.
Dazu könnte man jetzt in Kleinarbeit Exponent und Mantisse berechnen und wie oben beschrieben in 5 Adressen abspeichern. Doch wozu haben wir unseren Rechner, der das sowieso ständig tut?

Zwei wichtige ROM-Routinen helfen uns dabei:

INTFLP wandelt eine Integerzahl in eine Realzahl um und legt sie im FAC1 bereit. Wir kennen sie schon.

FACMEM legt den FAC 1 als reelle Zahl im Speicherformat ab.

Beispiel:
Die Zahl 10 soll als Realzahl ab 16500 gespeichert werden.

Teil 1:
Für die ROM-Routine **INTFLP** muß die Integerzahl im (A)-Register (HI der Integerzahl) und im (Y)-Register (LO der Integerzahl) bereitgestellt werden.

Teil 2:
Die ROM-Routine **FACMEM** benötigt im (X)-Register das LO-Byte der Anfangsadresse und im (Y)-Register das HI-Byte. Es werden dann 5 RAM-Adressen ab der Anfangsadresse überschrieben, die man sich natürlich merken muß, wenn man die Zahl wiederfinden will. Die Zahl 10 gibt es bereits im ROM. Wir können also überprüfen, ob die folgende Routine das gleiche Ergebnis liefert.

Teil 3:
Die Ausgabe einer reellen Zahl auf dem Bildschirm erledigen wir mit der betriebsinternen Routine **FLPOUT** (bereits erwähnt).

Die Startadressen dieser drei wichtigen ROM-Routinen:

Label	C64	40/80XX
INTFLP	45969=$b391	50364=$c4bc
FACMEM	48087=$bbd7	52493=$cd0d
FLPOUT	43708=$aabc	53133=$cf8d

Nun das ASSEMBLER-Programm, das die reelle Zahl 548 erzeugt, in (16500)...(16504) abspeichert und ausdruckt:

ASSEMBLER-Beispiel "14-facmem" (C64):

```
01 17200 lda #2      ; HI von 548
   17202 ldy #36     ; LO von 548
   17204 jsr 45969   ; INTFLP wandelt Integerzahl um
   17207 nop

02 17208 ldx #116    ; LO der Anfangsadresse
   17210 ldy #64     ; HI der Anfangsadresse
   17112 jsr 48087   ; FACMEM legt <FAC1> ab
   17215 nop

03 17216 jsr 43708   ; FLPOUT druckt: 548
   17219 rts
```

Ein Programm, das zum Rechnen diverse Konstanten benötigt, muß einen Vorspann haben, der diese Konstanten als reelle Zahlen in einen reservierten Bereich ablegt.

Wenn dabei Zahlen benötigt werden, die in den ROMs nicht enthalten sind, dann müssen diese Werte erst berechnet werden. Wie das funktioniert, schauen wir uns im nächsten Kapitel an.

4.3 Zahlenumwandlungen

4.3.1 Integer- in Realzahlformat mit INTFLP

Diese Umformung haben wir bereits kennengelernt. Siehe dazu Abschnitte 4.1.2 und 4.2.3.
Das Betriebssystem hält aber noch weitere Möglichkeiten bereit:

4.3.2 Reelle Zahl in Integerzahl mit FLPINT

Beispiel:
Die Zahl pi=3,1415... soll in eine Integerzahl umgewandelt werden. Die Nachkommastellen gehen dabei selbstverständlich verloren, sonst wäre das Ergebis keine Ganzzahl.

Vorbedingung:
Die Zahl pi steht im ROM (ab 44712 bzw. 48800/80XX)

Ablauf und ASSEMBLER-Beispiel "15-flpint" (C64):

```
01 lda #168       ; Anfangsadresse LO von pi
   ldy #174       ; dto. HI von pi
   jsr 48034      ; MEMFAC holt pi in den FAC1

02 jsr 47095      ; FLPINT wandelt pi in eine Integerzahl
```

Das Ergebnis dieser Umwandlung sind zwei Bytes (LO/HI), die in den Registern (Y/A) stehen. Zur Ausgabe mit **INTOUT** muß aber das LOW-Byte in (X) enthalten sein. Der Befehl TYX existiert aber nicht, so daß wir uns damit behelfen, die Zahl wieder in den FAC zu bringen und von dort auszugeben:

```
03 jsr 45969      ; INTFLP holt Ganzzahl aus (Y/A) nach FAC1
04 jsr 43708      ; FLPOUT gibt die Zahl aus: 3
   rts
```

4.3.3 Umwandlung eines Strings in eine reelle Zahl mit STRFAC

Mit Zeichenketten (Strings) kann man nicht rechnen, auch wenn sie Zahlenformat haben. Die Routine **STRFAC** formt einen String in eine reelle Zahl um und legt sie im FAC 1 bereit für weitere Operationen.
Dazu muß die Anfangsadresse des Strings in den Zeropageadressen (34/35) bzw. (31/32)/8OXX und die Länge des Strings im (A)-Register stehen.

Beispiel:
Die Zahl (-11O) soll über einen String erzeugt werden.
Als Stringanfang wählen wir die Adresse (256). Dort arbeitet nämlich auch das Betriebssystem mit solchen Umwandlungen.

Ablauf und ASSEMBLER-Beispiel "16-stringfac" (C64):

```
01 lda #49        ; Code für Ziffer 1
   sta 257        ; an 2. Stelle ablegen
   sta 258        ; an 3. Stelle ebenfalls 1 ablegen
   lda #48        ; Code für Ziffer O
   sta 259        ; an 3. Stelle ablegen
   lda #45        ; Code für Minuszeichen
   sta 256        ; an 1. Stelle setzen

02 lda #O
   sta 34         ; Anfangsadresse STRADR-LO
   lda #1
   sta 35         ; nach STRADR-HI
   lda #4         ; Länge des Strings nach (A)

03 jsr 47029      ; STRFAC bringt String als Wert nach FAC
04 jsr 43708      ; FLPOUT druckt:-11O
   rts
```

4.3.4 Umwandlung einer Zahl in einen String mit FLPSTR

Diese Operation wird in Abschnitt 6.7 ausführlich beschrieben. Hier deshalb nur kurz das Wichtigste:
Mit **FACSTR** wird eine Zahl, die im FAC1 steht, in eine Zeichenkette umgeformt, die ab Adresse (256) angelegt wird. Diese Startadresse steht dann in (A/Y).
Das Vorzeichen der Zahl steht dabei immer in (256), wobei positive Werte als erstes Zeichen ein Blank (Code 32) haben. Die erste Ziffer der Zahl steht also in 257.
Hat die Zahl einen kleineren Wert als 0.01, wird der String als Mantisse mit Vorzeichen und dem Exponent zur Basis 10 aufgebaut.

Beispiel:
Wir laden die Zahl pi aus dem ROM, wandeln sie in einen String um und geben 4 Nachkommastellen auf dem Bildschirm aus. Das entspricht dem BASIC-Befehl FRAC(pi,4), den die CBM-Rechner leider nicht kennen.

Ablauf und ASSEMBLER-Beispiel "17-flpstring" (C64):

```
01 17250 lda #168     ; LO von pi
   17252 ldy #174     ; HI von pi
   17254 jsr 48034    ; MEMFAC bringt pi nach FAC1

02 17257 jsr 48605    ; FLPSTR wandelt pi in String um

02 17260 ldx #0       ; Zähler auf Null setzen
   17262 inx          ; Schleifenanfang z. Suche nach Dezimalpunkt
   17263 lda 256,x    ; Zeichen aus dem String holen
   17266 cmp #46      ; Punkt ?
   17268 bne 17262    ; nein ---> weitersuchen

04 17270 ldy #0       ; Zähler für Nachkommastellen plus Punkt
   17272 lda 256,x    ; Nachkommastellen laden
   17275 sta 1424,x   ; und auf dem Bildschirm ausgeben
   17278 inx          ; Zähler für nächstes Zeichen
   17279 iny          ; und nächste Nachkommastelle erhöhen
   17280 cpy #5       ; letzte Stelle erreicht?
   17282 bne 17272    ; nein ---> weiterdrucken

05 17284  rts         ; ja  ---> fertig
```

Wenn Sie jetzt bei 17250 starten, wird ordnungsgemäß .1415 ausgegeben.

Aufgabe:
Bauen Sie diese Routine so um, daß auch das Vorzeichen mit ausgedruckt wird.

4.3.5 (A)-Inhalt in ASC-Code mit BYTHEX (nur 40/80XX)

BYTHEX wandelt den Inhalt des (A)-Registers in die Codezahl für das entsprechende HEX-Zeichen um.

Beispiel:
Welcher Hexzahl entspricht Bytewert 14?

Ablauf und ASSEMBLER-Beispiel "18-byte/hex" (80XX):

```
01 17350 lda #14      ; Bytewert 14 laden
02 17352 jsr 55098    ; BYTHEX erzeugt in (A) den HEX-Code
03 17355 sta 32768    ; Kontrollausdruck: E
   17358 rts
```

4.3.6 ASC-Code in entsprechendes Byte umwandeln mit HEXBYT

Das ist die Umkehrung zum oben besprochenen Fall 4.2.4.

Beispiel:
Suchen des Hexbyte-Codes für Zeichen "e", also ASCII-Code 69.

Ablauf und ASSEMBLER-Beispiel "19-hex/byte" (80XX):

```
01 17360 lda #69      ; Code für "e"
02 17362 jsr 55181    ; HEXBYT erzeugt Code für Hexzahl
03 17365 sta 32768    ; Kontrollausgabe (wie POKE32768,14): n
                        also war <A>=14
```

4.3.7 Positive Integerzahl in Realzahl mit ADRFLP

Im Gegensatz zur Routine **INTFLP** wird hier der Bereich von 0 bis 65535 verarbeitet. Vorzeichen werden also nicht erkannt. Die beiden Bytes der Integerzahl HI/LO müssen in (98/99) (!)

bzw. (95/96)/80XX gebracht werden. Außerdem wird im (X)-Register der Wert 144 verlangt und das Carry-Flag muß zur Vorbereitung gesetzt werden, bevor nach **ADRFLP** gesprungen wird.

Anmerkung: Der genannte Bereich ist der Adressbereich des Speichers. Deswegen der Label-Vorsatz ADR.

Beispiel:
Die Zahl 34048=$8500 soll in eine reelle Zahl verwandelt werden.

Ablauf und ASSEMBLER-Beispiel "20-adrflp" (C64):

```
01 17300 lda #0      ; LO
   17302 sta 99      ; nach FAC1+2
   17304 lda #133    ; HI
   17306 sta 98      ; nach FAC1+1

02 17308 ldx #144    ; Vorbereitung der Routine
   17310 sec

03 17311 jsr 48201   ; ADRFLP wandelt in reelle Zahl um

04 17314 jmp 43708   ; FLPOUT druckt: 34048
                       mit abschließendem Rücksprung
```

U m w a n d l u n g s - R O M - R o u t i n e n

L a b e l	C 6 4	4 0 / 8 0 X X
INTFLP	45969=$b391	50364=$c4bc
	positive/negative Integerzahl aus (Y/A) nach FAC1	
	Bereich: +/-32767	
ADRFLP	48201=$bc49	52607=$cd7f
	<98=$62> Integerzahl HI	<95=$5f>
	<99=$63> Integerzahl LO	<96=$60>
	<X>=144; SEC; positive Integerzahl nach FAC1	
	Bereich: 0 bis 65535 (Adressbereich)	
FLPINT	47095=$b7f7	51501=$c92d

<FAC 1>(reell)---> <FAC 1>(integer),LO/HI=(Y/A)

FLPSTR 48605=$bddd 53139=$cf93
<FAC 1> ---> Ziffernfolge ab (256), Ende=Byte 0

STRFAC 47029=$b7b5 51435=$c8eb
mit STRADR <34=$22/35=$23> Stringanfang <31=$1f/32=$20>
<A>=Stringlänge; Ergebnis in FAC1

BYTHEX ----- 55098=$d73a
<A>(Byte) ---> (A)(Hex-Byte)

HEXBYT ----- 55181=$d78d
<A>(ASC-Code) ---> (A)(Byte)

5

Arithmetik mit ROM-Routinen

5 Arithmetik mit ROM-Routinen

Die CBM-Rechner haben betriebseigene ROM-Routinen für die Durchführung arithmetischer Operationen mit reellen Zahlen.
Werden dabei zwei Zahlen verknüpft, so geschieht das mit Hilfe der beiden Floatpointakkus FAC1 und FAC2, die zunächst mit den gewünschten Zahlen geladen werden müssen, bevor die Rechenroutine angesprungen wird.

Sind die zu verknüpfenden Zahlen irgendwo im ROM oder RAM bereits als reelle Zahlen vorhanden, dann besteht auch die Möglichkeit, mit Hilfe der Anfangsadresse den FAC1 automatisch laden zu lassen, wenn man die entsprechende Einsprungstelle kennt.

Ist dies nicht der Fall, dann **muß** der FAC1 zuerst geladen werden. Erst vor dem Aufruf der Rechenroutine kann man - ebenfalls mit Hilfe der Anfangsadresse des Arguments - den FAC 2 belegen. Der Grund für diese Reihenfolge ist, daß beim Laden des FAC1 auch der FAC2 benützt wird und damit eventuell vorhandene Zahlen überschrieben werden.

Bei der Verwendung von LABELS werden wir ein "M" voranstellen, wenn damit die Einsprungadresse der ROM-Routine gemeint ist, die aus einem Speicher eine reelle Zahl in den FAC1 holt und anschließend verarbeitet ("M" wie MEMORY = Speicher).

Beispiel:
Die Routine **M-ADD** holt sich den zweiten Summanden aus der angegebenen Adresse und verknüpft ihn mit dem ersten Summanden. Zum Addieren mit der Routine **ADD** dagegen müssen die beiden Summanden in FAC1 und FAC2 vorher bereitgestellt werden.

Wird durch eine Operation ein- und dieselbe Zahl verändert (sog. monadische Operation im Gegesatz zu den dyadischen Operationen, bei denen zwei Zahlen verknüpft werden), so geschieht das im FAC1. Eventuell werden auch die Register in bestimmter Weise gesetzt.

Die folgenden Beispiele sind so einfach wie möglich gehalten und erläutern die eben genannten Vorgänge. Zur Übersicht finden Sie eine Label-Liste mit den Einsprungadressen für den C64 und die

40/80XX-Geräte. Für frühere Serien sollten Sie sich eine Referenzliste zulegen, die Sie bei fast allen handelsüblichen ROM-Listings finden.

Alle LABELS sind in der Form '**LABEL**' angeführt und bedeuten immer die Einsprungadressen für die mnemonisch abgekürzte Operation.

Bei den folgenden Beispielen wurde auf die Adressierung der ASSEMBLER-Befehle verzichtet, weil keine bedingten Sprünge in den Routinen vorkommen.

Als Vorbedingung wird in den meisten Beispielen eine reelle Zahl im RAM angenommen. Das bedeutet, daß Sie diese Zahl dort erst einmal erzeugen müssen, bevor Sie die angegebenen Routinen aufrufen. Wie das vor sich geht, haben wir in Abschnitt 4.2.3 mit Hilfe eines Zahlenbeispiels gelernt. Schauen Sie also im Zweifelsfall diesen Teil noch einmal gründlich durch.

5.1 Durch 10 dividieren mit FDIV10

Vorbedingung:
die reelle Zahl 36 befindet sich ab 16500 (HI=64/LO=116) im RAM.

Ablauf:
01 Die Anfangsadresse (LO/HI) wird nach (A/Y) geladen und durch **MEMFAC** in den FAC1 gebracht.
02 Aufruf der Routine **FDIV10.**
03 Wir speichern dann die neue Zahl 3.6 ab (16505) mit **FACMEM** ab, um sie später weiterverwenden zu können.
04 Zur Kontrolle erfolgt anschließend die Ausgabe auf dem Bildschirm, da durch FACMEM der FAC1 nicht verändert wird.

Aber Achtung:
Nach der Bildschirmausgabe mit **FLPOUT** ist der Inhalt des FAC1 zerstört!

ASSEMBLER-Programm "21-facdurch10" (C64):

```
01 lda #116          ; 16500=116/64, also
   ldy #64           ; Startadresse von Zahl 36 (LO/HI)
   jsr 48034         ; mit MEMFAC nach FAC1 bringen
02 jsr 47870         ; mit FDIV10 durch 10 dividieren
   ldx #121
   ldy #64           ; neue Zahl 3.6 mit
03 jsr 48087         ; FACMEM ab (16505)... abspeichern
04 jsr 43708         ; FLPOUT bringt 3.6 auf den Schirm
   rts
```

5.2 Mit 10 multiplizieren

Vorbedingung: wie oben
Ablauf: wie oben, aber Aufruf der Routine 'FMALL0'

ASSEMBLER-Programm "22-facmall0" (C64):

```
01 lda #116
   ldy #64           ; Zahl 36 ...
   jsr 48034         ; MEMFAC
02 jsr 47842         ; FMALL0
03 ldx #126
   ldy #64           ; neue Zahl 360 aus FAC1 nach (16510)...
   jsr 48087         ; mit FACMEM übertragen
04 jsr 43708         ; Ausgabe mit FLPOUT
   rts
```

5.3 Addieren des Werts 0,5 mit ADD0.5

Diese Routine wird benötigt, wenn Rundungen vorzunehmen sind.

Vorbedingung: wie oben
Ablauf: wie oben, aber Aufruf der Routine **ADD0.5**

ASSEMBLER-Programm "23-facplus0.5" (C64):

```
01 lda #116      ; Zahl 36
   ldy #64       ; nach
   jsr 48034     ; FAC1 mitMEMFAC
02 jsr 47177     ; ADD0.5
03 jsr 43708     ; FLPOUT bringt 36.5 auf den Bildschirm
   rts
```

5.4 Addieren beliebiger Zahlen mit ADD

Vorbedingungen:
Die beiden Summanden befinden sich im RAM oder ROM.
Beispiel: Zahl 36 ab (16500), Zahl 3.6 ab (16505).

Ablauf:
01 Zahl 36 nach FAC1
02 Zahl 36 nach FAC2 mit **MEMFC2**
03 Aufruf der Routine **ADD**
04 Bildschirmausgabe zur Kontrolle

ASSEMBLER-Programm "24-addieren" (C64):

```
01 lda #121
   ldy #64
   jsr 48034       ; 3.6 nach FAC1 mit MEMFAC
02 lda #116
   ldy #64
   jsr 47756       ; 36 nach FAC2 mit MEMFC2
03 jsr 47210       ; ADD
04 jsr 43708       ; Bildschirmausgabe: 39.6
   rts
```

5.5 Addieren beliebiger Zahlen mit M-ADD

Der Unterschied zu **ADD** ist, daß FAC2 selbständig geladen wird, wenn die Anfangsadresse des 2. Summanden in (A/Y) mit LO/HI steht.

Vorbedingungen: wie 5.4
Ablauf: zunächst wie oben

M-ADD ersetzt die beiden Routinen **MEMFC2** und **ADD.**

ASSEMBLER-Programm "25-mem-addition" (C64):

```
01 lda #121
   ldy #64          ; 3.6 nach
   jsr 48034        ; mit MEMFAC
02 lda #116
   ldy #64
03 jsr 47207        ; M-ADD rechnet 36+3.6
04 jsr 43708        ; FLPOUT ---> Bildschirmausgabe: 39.6
   rts
```

Die folgenden Routinen zum Subtrahieren, Multiplizieren und Dividieren sowie zum Potenzieren/Radizieren können ebenfalls auf zwei Arten durchgeführt werden.
Wir stellen in den folgenden Beispielen aber jeweils nur eine Variante dar. Die ENTRY-Points für die zweite Möglichkeit finden Sie in der Label-Liste.

5.6 Subtrahieren mit M-SUB

Vorbedingungen:
Minuend und Subtrahend im RAM oder ROM.

Ablauf:
01 Subtrahend nach FAC 1
02 Anfangsadresse LO/HI des Minuenden nach (A/Y)
03 Aufruf von **M-SUB**
...

ASSEMBLER-Beispiel "26-mem-subtra" (C64):

```
01 lda #121
   ldy #64          ; 3.6 nach
   jsr 48034        ; mit MEMFAC
```

```
02 lda #116
   ldy #64          ; Anfangsadresse von 36 nach (A/Y)
03 jsr 47184        ; M-SUB rechnet 36-3.6
04 jsr 43708        ; Bildschirmausgabe: 32.4
   rts
```

Bei manchen Rechenvorgängen kommt es vor, daß man im FAC1 schon den Minuend hat. Subtrahiert man nun mit **M-SUB**, dann erhält man den Differenzwert mit dem umgekehrten Vorzeichen.

Mit der folgenden Routine kehrt man es wieder um.

5.7 Vorzeichenwechsel mit FACMIN

Vorbedingung:
Zahl im FAC1. Im Beispiel Zahl 36.

Ablauf:
01 Zahl in den FAC laden
02 Aufruf von **FACMIN**
03 Abspeichern der negativen Zahl (-36 wird im nächsten Beispiel wieder verwendet)

ASSEMBLER-Beispiel "27-vorz-wechsel" (C64):

```
01 lda #116
   ldy #64        ; 36 nach
   jsr 48034      ; FAC1 mit MEMFAC
02 jsr 49076      ; FMINUS ändert das Vorzeichen im FAC1
03 ldx #141
   ldy #64        ;  -36  nach (16525)...
   jsr 48087      ; ... abspeichern mit FACMEM
04 jsr 43708      ; Bildschirmausgabe mit FLPOUT : -36
   rts
```

5.8 Betrag einer Zahl mit FACABS

Vorbedingung:
Positive oder negative reelle Zahl steht im FAC1 (Beispiel: -36).

Ablauf:
01 Zahl -36 in FAC1 laden
02 Aufruf von **FACABS** erzeugt immer ein positives Vorzeichen (absoluter Betrag)
...

ASSEMBLER-Beispiel "28-abs.betrag" (C64):

```
01 lda #141
   ldy #64          ; -36 nach
   jsr 48034        ; FAC1 mit MEMFAC
02 jsr 48216        ; FACABS erzeugt positives Vorzeichen
03 jsr 43708        ; Bildschirmausgabe mit FLPOUT: 36
   rts
```

5.9 Multiplizieren mit M-MULT

Dies entspricht in Vorbedingungen und Ablauf dem Addieren.

ASSEMBLER-Beispiel "29-mem-mult" (80XX):

```
01 lda #116
   ldy #64          ; 36 nach
   jsr 48034        ; FAC1 mit MEMFAC
02 lda #121
   ldy #64          ; 3.6 nach
   jsr 47656        ; M-MULT multipliziert MEM mit FAC1
03 jsr 43708        ; Bildschirmausgabe mit FLPOUT: 129.6
   rts
```

5.10 Division mit M-DIV

Vorbedingung:
Dividend und Divisor im ROM oder RAM.

Ablauf:
01 Divisor nach FAC1
02 Anfangsadresse des Dividenden nach (A/Y)
03 Aufruf von **M-DIV**
...

ASSEMBLER-Beispiel "30-mem-division" (C64):

```
01 lda #116
   ldy #64        ; 36 nach FAC1 (Divisor!)
   jsr 48034      ; mit MEMFAC
02 lda #121
   ldy #64        ; Adresse von 3.6 nach (A/Y) (Dividend!)
03 jsr 47887      ; M-DIV teilt 3.6 durch 36
04 jsr 43708      ; Bildschirmausgabe mit FLPOUT: .1
   rts
```

Dies entspricht im Ablauf der Subtraktion.
Auch hier hat man oft den Dividend schon im FAC1. Am besten führt man nun die Division umgekehrt durch und bildet danach den Kehrwert.
Die folgende Routine ist für solche Fälle geeignet.

5.11 Kehrwert bilden mit M-DIV

Vorbedingung:
Zahl 1 steht im ROM (ist bei allen CBM-Rechnern erfüllt).

Ablauf:
01 Zahl in den FAC1 holen, z.B. 10 aus dem ROM
02 Adresse von 1 nach (A/Y)
03 Aufruf von **M-DIV**
...

ASSEMBLER-Beispiel "31-kehrwert" (C64):

```
01 lda #249      ; LO
   ldy #186      ; HI (Anfangsadresse von 10)
   jsr 48034     ; 10 nach FAC1 mit MEMFAC
02 lda #188      ; LO
   ldy #185      ; HI (Anfangsadresse von 1)
03 jsr 47887     ; M-DIV teilt 1 durch 10
04 jsr 43708     ; Bildschirmausgabe mit FLPOUT: .1
   rts
```

5.12 Quadratwurzel ziehen mit SQRFAC

Dies ist eigentlich eine Sonderform des Potenzierens, das im nächsten Abschnitt besprochen wird.
(Der Exponent ist eben beim Quadratwurzelziehen 0.5)

Vorbedingung:
Radikand im RAM oder ROM.

Ablauf:
01 Radikand in den FAC1 laden
02 Aufruf von **SQRFAC**
...

ASSEMBLER-Beispiel "32-wurzel" (C64):

```
01 lda #116
   ldy #64        ; 36 nach
   jsr 48034      ; FAC1 mit MEMFAC
02 jsr 49009      ; SQRFAC zieht Quadratwurzel aus FAC1
03 jsr 43708      ; FAC1 auf Schirm mit FLPOUT : 6
   rts
```

5.13 Potenzieren und Radizieren mit POTRAD

Ist der Exponent eine natürliche Zahl, erfolgt echtes Potenzieren (als vereinfachte Form der Mehrfachmultiplikation mit gleichen Faktoren).
Bei Exponenten der Form 1/n, mit n = natürliche Zahl ist der Potenzwert die n-te Wurzel aus der Basis.
Negative Exponenten erzeugen zusätzlich noch den Kehrwert.

Vorbedingungen:
Die Basis steht im RAM oder ROM.
(Im Beispiel wird der Exponent erzeugt.)

Ablauf:
01 Laden der Basis in den FAC2
02 Der Exponent (hier 4) wird als Integerzahl erzeugt und als reelle Zahl in den FAC 1 gebracht mit **INTFLP**.
03 Aufruf der Routine **POTRAD**
...

ASSEMBLER-Beipsiel "33-potenzieren" (C64):

```
01 lda #116
   ldy #64        ; 36 nach FAC2
   jsr 47756      ; mit MEMFC2
02 lda #0
   ldy #4         ; Zahl 4 nach FAC1
   jsr 45969      ; mit INTFLP
03 jsr 49019      ; POTRAD rechnet 36 hoch 4
04 jsr 43708      ; Bildschirmausgabe mit FLPOUT : 1679616
   rts
```

Sind Basis u n d Exponent im ROM oder RAM vorhanden, lädt man zuerst den Exponenten in den FAC1, setzt (A/Y) auf die Anfangsadresse der Basis und ruft dann **M-POT** auf.
Der Potenzwert steht dann wieder im FAC 1.

Probieren Sie mal die Kehrwertbildung aus, indem Sie nach **INTFLP** (jsr 45969) noch **FMINUS** aufrufen, bevor potenziert wird.

Auf diese Weise läßt sich auch der Kehrwert einer beliebigen Zahl bilden, wenn Sie als Exponent (-1) wählen.

5.14 Logarithmieren mit LOGNAT

Alle Logarithmen beziehen sich hier auf die Basis e. Wir rechnen also mit dem Logarithmus naturalis.

Vorbedingung:
Der Numerus kann erzeugt werden oder im RAM oder ROM stehen.

Ablauf:
01 Erzeugen der Zahl 1000 im FAC1
02 Aufruf von **LOGNAT**
03 Ablegen von ln(1000) ab (16515)
...

ASSEMBLER-Beispiel "34-log nat" (C64):

```
01 lda #3
   ldy #232      ; Zahl 1000 nach
   jsr 45969     ; FAC1 mit INTFLP
02 jsr 47594     ; LOGNAT bildet Logarithmus von 1000
03 ldx #131
   ldy #64       ; ab (16515)
   jsr 48087     ; mit FACMEM das Eregebnis ablegen
04 jsr 43708     ; Bildschirmausgabe mit FLPOUT : 6.90775528
   rts
```

5.15 Exponentialrechnen mit EHOCHF

Mit der folgenden Routine führen wir die Umkehrung zum Logarithmieren durch und haben gleichzeitig eine Probe, ob wir richtig programmiert haben.

Vorbedingung:
Logarithmus steht im RAM oder ROM.

Ablauf:
01 Logarithmus in den FAC1 bringen.
02 Aufruf von **EHOCHF**
...

ASSEMBLER-Beispiel "35-ehochfac" (C64):

```
01 lda #131
   ldy #64       ; ln1000  nach
   jsr 48034     ; FAC1 bringen mit MEMFAC
02 jsr 49133     ; EHOCHF rechnet e hoch (ln 1000)
03 ldx #136
   ldy #64       ; 1000 wird ab (16520)
   jsr 48087     ; abgelegt mit FACMEM
04 jsr 43708     ; Bildschirmausgabe mit FLPOUT : 1000
   rts
```

5.16 Erzeugen einer Zufallszahl mit ZUFALL

Zufallszahlen werden im Bereich von 0 bis 1 mit der Routine **ZUFALL** erzeugt und im FAC1 abgelegt.
Damit es uns nicht so langweilig wird, stellen wir uns das Problem, daß die Zufallszahl eine natürliche Zahl aus dem Bereich von 0 bis 36 sein soll.
Und dieses Ergebnis soll Integer-Format haben.

Vorbedingung:
Zahl 36 ist im RAM oder ROM vorhanden.

Ablauf:
01 Erzeugen einer Zufallszahl mit **ZUFALL** liefert einen Wert im FAC1 zwischen 0 und 1.
02 Wir multiplizieren diesen Wert mit 36.
03 ... und addieren 0.5 wegen der folgenden Rundung
04 durch Umwandlung in eine Integerzahl.
Diese Integerzahl steht nun mit LO/HI in (Y/A).
Uns interessiert zwar im Moment nur das LO-Byte, wir halten aber die Möglichkeit für größere Zahlen als 255 offen und schleppen das HI-Byte zur Übung mit.
05 Zur Bildschirmausgabe mit **INTOUT** muß aber das LO-Byte in (X) und das HI-Byte in (Y) stehen.
Den Befehl TYX (transportiere <Y> nach <X>) gibt es leider nicht.
Wir retten daher den Akku-Inhalt <A> auf den Stack, holen <Y> nach (A) und können von hier aus <A> nach (X) übertragen.

06 Nun holen wir den ursprünglichen Akku-Inhalt vom Stapel und setzen ihn in (Y) ein.
07 Jetzt gibt die Routine **INTOUT** die richtige Zahl aus.

ASSEMBLER-BEISPIEL "36-zufall(x)" (C64):

```
01 jsr 57495     ; ZUFALL holt Zufallszahl nach FAC1
02 lda #116
   ldy #64       ; Zahl 36 mit FAC1
   jsr 47656     ; multiplizieren mit M-MULT
03 jsr 47177     ; ADDO.5 addiert 0.5 zu <FAC1>
04 jsr 47095     ; FLPINT wandelt FAC in Integer
                   gleichzeitig steht das Ergebnis LO/HI in (Y/A)
05 pha           ; HI-Byte auf Stack retten
   tya           ; LO-Byte von (Y) nach (A) übertragen
   tax           ; LO-Byte nach (X) bringen
06 pla           ; HI-Byte vom Stack nehmen
   tay           ; ...und nach (Y) übertragen
07 jsr 48589     ; Bildschirmausgabe mit INTOUT: RND(36)
   rts
```

5.17 Winkelfunktionen mit SINUS, COSIN und TANG

Die interne Rechenoperation mit Winkeln erfolgt immer in der Einheit RAD.
Zur Umrechnung gilt: $2pi(RAD) = 360^{o}(DEGREE)$

Im folgenden Beispiel gehen wir davon aus, daß die Gradzahl zunächst in DEGREE vorliegt.
Da die ROM-Routinen für die Winkelfunktionen ansonsten sehr einfach zu handhaben sind, geben wir hintereinander die drei Hauptfunktionswerte aus.
Dazu speichern wir den einmal ermittelten Winkel in der Einheit RAD ab 16530 ab.

Aufgabe:
Berechnen Sie die Funktionswerte: $sin36^{o}$, $cos36^{o}$ und $tan36^{o}$ und geben Sie sie auf dem Bildschirm aus.

Vorbedingungen:
Die Zahlen 360 und 36 sind im RAM.
Zahl 2pi liegt im ROM (C64:58121; 40/80XX:54055)

Ablauf:
- 360 nach FAC1 bringen
- 36 durch 360 teilen ...
- ... und mit 2pi multiplizieren
- ... und ab (16530) speichern. Damit steht der Winkel im Bogenmaß für die weiteren Operationen zur Verfügung.
- Aufruf **SINUS** und Bildschirmausgabe
- Winkel wieder nach FAC1, Aufruf **COSIN**...
- usw.

ASSEMBLER-BEISPIEL "37-winkelfunktn" (C64):

```
- lda #126
  ldy #64        ; 360 aus (16510...) nach
  jsr 48034      ; FAC1 mit MEMFAC
- lda #116
  ldy #64        ; Anfangsadresse von 36 ist (16500)
  jsr 47887      ; M-DIV teilt 36 durch 360
- lda #9
  ldy #227       ; Anfangsadresse (54055) von 2pi
  jsr 47656      ; M-MULT' rechnet FAC 1 mal 2pi
- ldx #146
  ldy #64        ; ab (16530)... wird nun
  jsr 48087      ; mit FACMEM der Winkel in RAD abgelegt
- jsr 57963      ; SINUS berechnet sin36°
- jsr 43708      ; FLPOUT:  0.587785252

- lda #146
  ldy #64        ; Winkel holen
  jsr 48034      ; mit MEMFAC
- jsr 57956      ; COSINUS berechnet cos36°
- jsr 43708      ; FLPOUT: 0.809016994

- lda #146
  ldy #64
  jsr 48034      ; Winkel wieder nach FAC1
- jsr 58036      ; TANGENS berechnet tan36°
- jsr 43708      ; FLPOUT: 0.726542528
  rts
```

5.18 Umkehrung der Winkelfunktionen mit ARCTAN

Zur Winkelbestimmung aus den trigonometrischen Funktionswerten sin, cos, tan, cot existiert in den ROM-Routinen der CBM-Rechner nur die Umkehrfunktion **ARCTAN.**
Nach Aufruf dieser Routine enthält der FAC 1 den Winkel im Bogenmaß.
In unserem Beispiel nehmen wir wieder die Umrechnung in DEG vor. Dazu wollen wir arctan(0.5) berechnen und die gefundene Gradzahl in DEG ausgeben.

Vorbedingung:
0.5 steht im ROM (C64:ab(48913);40/80XX:53477).
2pi und 1 stehen ebenfalls im ROM (siehe 5.17).
360 steht im RAM bei (16510)...

Ablauf:
- Argument (0.5) in den FAC 1 bringen
- Aufruf von 'ARCTAN': berechnet Winkel in RAD
- Multiplikation mit 2pi und Kehrwertbildung
- Multiplikation mit 360
  Damit erfolgt die Umrechnung in DEGREE:
  x=360·arctan0.5/2pi
- Ablegen des Winkels für spätere Verwendung
- Ausgabe auf dem Bildschirm

ASSEMBLER-Beispiel "38-arcus tan" (C64):

```
- lda #17
  ldy #191      ; Argument 0.5 nach
  jsr 48034     ; FAC1 mit MEMFAC
- jsr 58126     ; ARCTAN rechnet arctan 0.5 in RAD
- lda #9
  ldy #227      ; 2pi (Anfangsadresse)
  jsr 47887     ; M-DIV rechnet 2pi/arctan 0.5
  lda #188
  ldy #185      ; 1 (Anfangsadresse)
  jsr 47887     ; M-DIV bildet Kehrwert
- lda #126
  ldy #64       ; 360 (aus RAM)
  jsr 47656     ; M-MULT multipliziert FAC1 mit 360
```

```
- ldx #151
  ldy #64       ; Speicheranfang (16535) festlegen
  jsr 48087     ; und Winkel in DEGREE mit FACMEM ablegen
- jsr 43708     ; FLPOUT: 26.5650512
  rts
```

5.19 Weitere Arcus-Funktionen mit ARCTAN

Zur Berechnung von arcsin(x) ist die Formel anzuwenden:
$\arcsin(x)=\arctan(x/\sqrt{1-x^2})$
Das folgende Beispiel berechnet den Winkel zu arcsin(0.5).

Vorbedingung:
wie 5.18

Ablauf:
- Argument (hier 0.5) nach FAC1 und mit sich selbst multiplizieren. Die Potenzierroutine lohnt sich hier nicht.
- Berechnen von $(1-0.5^2)$
- Quadratwurzelziehen aus $(1-0.5^2)$
- Argument 0.5 wieder holen und durch den Wert im FAC1 teilen
- Aufruf von **ARCTAN**.
  Damit steht arcsin(0.5) im FAC1, allerdings im Bogenmaß.
  (Wenn Sie wollen, überprüfen Sie das mit JSR 43708.)
- Zur Umrechnung in DEGREE können wir den letzten Teil der in Abschnitt 5.18 besprochenen Routine verwenden. Dazu springen wir die Adresse an, bei der die Umrechnung in die üblichen Gradzahlen beginnt, also den dritten Punkt.

ASSEMBLER-Beipsiel "39-arcus sinus" (C64):

```
- lda #17
  ldy #191      ; Argument 0.5
  jsr 48034     ; nach FAC1 mit MEMFAC
- lda #17
  ldy #191      ; Anfangsadresse von 0.5
  jsr 47656     ; M-MULT rechnet 0.5 mal 0.5
- lda #188
  ldy #185      ; Anfangsadresse von 1 (ROM)
  jsr 47184     ; M-SUB rechnet 1-0.5^2
```

```
- jsr 49009     ; SQRFAC zieht Quadratwurzel aus <FAC1>
  lda #17
  ldy #191      ; Anfangsadresse 0.5 (ROM) setzen
  jsr 47887     ; M-DIV teilt 0.5/<FAC1>
- jsr 58126     ; ARCTAN berechnet Winkel in RAD
- ... Routine zur Umrechnung in DEGREE und Bildschirmausgabe:
  als Ergebnis müßte 30 herauskommen.
  rts
```

Aufgabe:
Erstellen Sie ein ASSEMBLER-Programm für die Berechnung von arccos.
Die Formel dafür lautet: **arccos(x)=arctan(sqr(1-x^2)/x)**

5.20 Polynomauswertung mit POLNOM

Die Betriebsroutine kann Polynome bis zum 255. Grad berechnen, wenn in einer Tabelle im ROM oder RAM vorhanden sind:

a) der Grad des Polynoms
b) die Konstanten beginnend mit dem Koeffizienten der höchsten Potenz

Die Anzahl der Konstanten ist dabei immer um eins höher als der Polynomgrad, weil das letzte Glied eines Polynoms immer das variablenfreie Glied ist.

Beispiel:

$a_4x^4+a_3x^3+a_2x^2+a_1x+a_0$

ist ein Polynom des 4. Grades mit den Konstanten a_4 bis a_0. a_0 ist das variablenfreie Glied.

Zahlenbeispiel:
$10x^2+0.25x+10$ ist ein quadratisches Polynom (2. Grad) mit den Koeffizienten: $a_2=10$; $a_1=0.25$; $a_0=10$

Ablauf und ASSEMBLER-Beispiel "40-polykon" (C64):

Bleiben wir bei dem Zahlenbeispiel und legen die Konstanten in einer Tabelle an, die z.B. bei (16640)=(0/65) beginnt.
Dann muß in der ersten Adresse (16640) der Grad des Polynoms als Byte stehen.
Das geschieht mit

```
01 lda #2
   sta 16640
```

02 Anschließend werden die Konstanten mit Hilfe der Routine **FACMEM** als reelle Zahlen abgelegt. Da jede relle Zahl fünf Adressen beansprucht, geschieht das in 5-er Abständen:

```
   lda #249
   ldy #186     ; a2=10 (aus dem ROM)
   jsr 48034    ; in den FAC1 mit MEMFAC
   ldx #1
   ldy #65      ; Anfangsadresse (16641) setzen
   jsr 48087    ; FACMEM legt reelle Zahl ab
```

```
03 ldx #11
   ldy #65      ; 10 ist noch im FAC1 und kann als a0 nach
   jsr 48087    ; (16651...) mit FACMEM gebracht
```

04 Fehlt nur noch die Konstante a_1=0.25, die nach (16646...) gesetzt werden muß:

```
   lda #234
   ldy #226     ; Anfangsadresse von 0.25 im ROM ist (58090)
   jsr 48034    ; Argument 0.25 nach FAC1 mit MEMFAC
   ldx #6       ; Anfangsadresse (16646)
   ldy #65      ; nach (X/Y)
   jsr 48087    ; FACMEM legt 0.25 ab
   rts
```

Kommen wir zur eigentlichen Polynom-Auswertung:

Vorbedingungen:
Polynomgrad und Konstanten müssen lückenlos im RAM oder ROM sein. (Die eben besprochene Laderoutine muß aufgerufen worden sein.) Die Anfangsadresse dieser Tabelle muß bekannt sein (hier 16640).

Ablauf:
01 Die Belegung für x wird als reelle Zahl in den FAC1 gebracht. In unserem Beispiel ist das Argument -0.5 (steht im ROM).

02 Die Register (A/Y) müssen mit LO/HI der Anfangsadresse unserer Konstantentabelle belegt werden.
03 Aufruf der Polynomauswertung
04 Bildschirmausgabe

ASSEMBLER-Beispiel "41-poly-wert" (C64):

```
01 lda #224     ; (47584) ist
   ldy #185     ; Anfangsadresse von (-0.5) aus dem ROM
   jsr 48034    ; MEMFAC
02 lda #0       ; Beginn der Tabelle LO
   ldy #65      ; Beginn der Tabelle HI, also (16640)
03 jsr 57433    ; POLNOM berechnet Polynomwert
                   für die Belegung x:=-0.5
04 jsr 43708    ; FLPOUT: 12.375
```

Prüfen wir das nach und nennen wir den Polynomwert y:
$y=10x^2+0.25x+10$ mit x:=-0.5 ===>
$y=10(-0.5)(-0.5)+0.25(-0.5)+10$
$y=2.5-0.125+10$
$y=12.375$

Aufgaben:
Lassen Sie sich zur Übung die Polynomwerte für andere Argumente ausrechnen.

Erstellen Sie ein Programm zur Berechnung von Polynomen höheren Grades.

5.21 Wertetabellen für Funktionen mit POLNOM

Das Polynom-Beispiel aus 5.20 ist nichts anderes als der Funktionsterm einer Parabelgleichung.
Es lassen sich auf diese Weise selbstverständlich die Funktionswerte für Terme höheren Grades berechnen.

Zum Anlegen einer Wertetabelle gehen wir von der kleinsten Belegung in festgelegten Schritten bis zum gewünschten größten Argument und rufen jeweils die **POLNOM**-Routine auf.

Nehmen wir an, das Intervall für x sei -5<x<5 und die Schrittweite sei 0.5. Dann werden insgesamt 21 Werte gesucht.

Vorbedingungen:
- Die zugehörige Konstantentabelle steht ab (16640)...(wie 5.20).
- Für das Argument (die Belegung) reservieren wir 5 Adressen ab (16530) bis (16534).
- Das ASSEMBLER-Beispiel aus 5.20 verändern wir dahingehend, daß wir die Anfangsadresse (16530)=(146/64) des variablen Arguments nach (A/Y) zum Vorbereiten der **MEMFAC**-Routine laden müssen.
- Ersetzen wir also in Teil 01 von "polywert"
  lda #224 durch lda #146 und
  ldy #185 durch ldy #64,
  dann können wir "polywert" als Programmteil übernehmen.
- Als Zähler für die Begrenzung des Intervalls benötigen wir eine freie Adresse: (1000).

Ablauf:

Teil 1: Initialisierungen
- Anfangswert des Zählers +1 setzen: <1000>=22
  Die Erhöhung des Zählers um eine Einheit ist deshalb notwendig, weil die Hauptschleife, in der jeweils der nächste Wert bestimmt wird, mit dem Befehl DEC 1000, also einer Erniedrigung des Zählers beginnt.
- Erzeugen der Zahl -5 im FAC1 als Wert für die erste Belegung.

Teil 2: Rechen- und Ausdruck-Schleife
In einer Schleife wird jeweils
- der Zähler um 1 erniedrigt und auf 0(=Ende) geprüft
- das Argument ab (16530) bereitgestellt und auf dem Bildschirm ausgegeben,
- ein Trennzeichen ausgedruckt,
- die Polynomroutine aufgerufen,
  der Polynomwert ausgedruckt,
- ein Line-Feed ausgegeben,
- das Argument um 0.5 erhöht
- und zum Schleifenanfang verzweigt.

Zur Ausgabe auf dem Bildschirm verwenden wir vorab ohne nähere Erklärungen die ROM-Routine **CHROUT**. Sie gibt das Byte aus, das im (A)-Register steht. Das entspricht dem BASIC-Befehl PRINT CHR$(A). Näheres dazu im nächsten Kapitel "Ausgabe-Routinen".

ASSEMBLER-Beispiel "42-wertetabelle" (C64):

```
Teil 1: Anfangswerte
- 19550 lda #0       ; HI
  19552 ldy #5       ; LO
  19554 jsr 45969    ; INTFLP wandelt 5 in reelle Zahl
  19557 jsr 49076    ; FACMIN ändert Vorzeichen ===> -5 im FAC1
  19560 lda #22      ; Schrittanzahl + 1 festlegen
  19562 sta 1000     ; und im Zähler (1000) als Erstwert ablegen

Teil 2: Hauptschleife
- 19565 dec 1000     ; Zähler erniedrigen
  19568 beq 19624    ; mit 0 vergleichen; bei 0 Sprung ans Ende

- 19570 ldx #146     ; LO Anfangsadresse
  19572 ldy #64      ; HI des Arguments ab (16530)...
  19574 jsr 48087    ; FACMEM legt Argument bereit

- 19577 ldy #1       ; (Y)-Belegung für Ausgabe von negativer Zahl
- 19579 jsr 48599    ; mit FPOUTX gibt Argument aus

- 19582 lda #47      ; Code für "/"
  19584 jsr 65490    ; CHROUT erzeugt auf den Bildschirm: /

"polywert": ----------------------------------------------
- 19587 lda #146
  19589 ldy #64      ; aktuelles Argument ...
  19591 jsr 48034    ; ... nach FAC1 mit MEMFAC
- 19594 lda #0       ; LO und
  19596 ldy #65      ; HI der Adresse der Konstantentabelle
  19598 jsr 57433    ; POLNOM berechnet Polynomwert mit ak-
                       tueller Belegung
- 19601 ldy #1       ; (beliebige Zahl ausgeben)
  19603 jsr 48599    ; mit FLPOUT gibt berechneten Wert aus
-------------------------------------------------------------
- 19606 lda #13      ; Code für "RETURN"
  19608 jsr 65490    ; CHROUT gibt CR/Linefeed aus

- 19611 lda #146
  19613 ldy #64      ; altes Argument nach FAC1 mit
  19615 jsr 48034    ; MEMFAC
  19618 jsr 47177    ; ADDO.5 addiert 0.5 zum FAC1
```

```
- 19621 clc
  19622 bcc 19565    ; erzwungener Sprung zum Schleifenanfang

- 19624 rts
```

Anmerkung:
Im ersten Teil von "polywert" muß noch einmal das Argument in den FAC geladen werden, da nach einer Ausgabe mit **FLPOUT** der Inhalt des FAC1 zerstört ist.

5.22 Vergleichsoperationen mit CMPFAC

Die Routine **CMPFAC** vergleicht den Inhalt von FAC1 mit einer reellen Zahl, die im Speicher steht.
Dazu muß zunächst der FAC1 geladen werden und anschließend die Anfangsadresse der Vergleichszahl nach (A/Y) gebracht werden.

Der Vergleich hat immer die Form FAC1 ? MEMORY .
Als Ergebnis dieses Vergleichs ist das (A)-Register wie folgt belegt:

```
FAC1 > MEMORY  ===>  <A>=  1
FAC1 = MEMORY  ===>  <A>=  0
FAC1 < MEMORY  ===>  <A>=255
```

Beispiel: Wir vergleichen die Zahlen 360 und 36, die wir schon mehrfach benützt haben und geben das entsprechende Zeichen (>,= oder <) in der unteren rechten Bildschirmecke aus.

Vorbedingungen:
- Die Zahl 360 steht ab (16510),
- die Zahl 36 ab (16500) im RAM.

Ablauf:
01 Zahl 360 in FAC1 laden
02 Anfangsadresse von 36 nach (A/Y) bringen
03 **CMPFAC** aufrufen
04 bei Ergebnis 0 verzweigen und "=" ausgeben, dann Ende.
05 sonst ROL im (A)-Register durchführen
06 <C-Flag> jetzt 0, dann verzweigen und ">" ausgeben, dann Ende
07 <C-Flag> sonst 1, dann "<" ausgeben und Ende.

Erläuterung der Bit-Operation ROL (Rotation nach links):

Hier dreht es sich nur noch darum festzustellen, ob <A> =255 oder <A>=1 ist, denn <A>=0 wurde schon geprüft.

a) bei <A>=255 stehen im (A)-Register die Bits: 11111111
   Der Befehl ROL schiebt nun alle Bits um eine Stelle nach links und setzt das auf der linken Seite hinausgeschobene Bit in das (C)-Register. In diesem Fall wird das Carry-Flag also gesetzt.

b) Bei <A>=1 stehen die Bits: 00000001.
   ROL schiebt also eine Null in das (C)-Register, löscht also das Carry-Flag.

Wir brauchen deshalb nach dem ROL-Befehl nur noch mit BCC überprüfen, in welchem Zustand sich das (C)-Flag befindet und ersparen uns somit die umständlicheren CMP-Befehle.

ASSEMBLER=Beispiel "43-vergleich" (C64):

```
01 19650 lda #126
   19652 ldy #64       ; 360 nach
   19654 jsr 48034     ; FAC1 mit MEMFAC
02 19657 lda #116      ; Anfangsadresse LO von 36
   19659 ldy #64       ; Anfangsadresse HI von 36
03 19661 jsr 48219     ; CMPFAC vergleicht: 360 ? 36

04 19664 beq 19677     ; letzte Operation 0 (also Gleichheit)?
                         ja ===> Sprung
05 19666 rol           ; Linksrotation in (A)
06 19667 bcc 19673     ; C-Flag gelöscht (also größer) ===> Sprung
07 19669 ldx #60       ; sonst Code für "<" laden und
   19671 bne 19679     ; zum Ausgabeteil springen

06 19673 ldx #62       ; Code für ">" laden und
   19675 bne 19679     ; zum Ausgabeteil springen

   19677 ldx #61       ; Code für "=" laden

08 19679 stx 2023      ; Bildschirmausgabe und
   19682 rts           ; Ende
```

Vertauschen Sie einmal die Anfangsadressen von 360 und 36 oder geben Sie beide Male die gleichen LO/HIs ein.
Sie sehen, unsere Routine eignet sich nicht nur für die beiden Zahlen 360 und 36, sondern für alle reellen Zahlen.

Aufgabe:
Wandeln Sie das ASSEMBLER=Beispiel so ab, daß Sie einen kompletten Vergleichsausdruck (wahre Aussage) auf dem Bildschirm ausgeben können.

Anwendungsmöglichkeiten:
Bei Rechenoperationen muß in einigen Fällen vorher geprüft werden, ob z.B. eine Zahl den Wert 0 hat. Ist dies der Fall, dann darf z.B. nicht durch diese Zahl dividiert werden, sonst erscheint "division by zero error".

5.23 Vorzeichenprüfung mit SGNFAC

Für Winkelberechnungen im Bereich von 0 bis 360^0 sind z.B. Vorzeichenprüfungen vorzunehmen, um den Winkel mit Hilfe der trigonometrischen Funktionen eindeutig angeben zu können.
Hat man zum Beispiel den Sinuswert und den Cosinuswert eines Winkels, dann läßt sich dieser im Bereich bis 360^0 eindeutig angeben, was mit einem einzigen Funktionswert ja nicht der Fall ist (zum Sinuswert 0.5 lassen sich die Winkel 30^0 aber auch 150^0 zuordnen).

Folgende Regeln aus der Schulmathematik sollten Sie sich noch einmal in Erinnerung rufen:

Vorzeichen SINUS	Vorzeichen COSINUS	Winkel zwischen
plus	plus	0 und 90
plus	minus	90 und 180
minus	minus	180 und 270
minus	plus	270 und 360

Doch zurück zu unserer ROM-Routine:
Die Routine **SGNFAC** prüft das Vorzeichen im FAC1 und setzt das (A)-Register folgendermaßen:

Zahl negativ ===> <A> = 255
Zahl = 0 ===> <A> = 0
Zahl positiv ===> <A> = 1

Vorbedingung:
Die zu prüfende Zahl muß im FAC1 stehen oder zuerst dorthin ge-

bracht werden.

Ablauf:
01 Zahl in den FAC1 übertragen
02 Aufruf von **SGNFAC**
03 Überprüfung des (A)-Registers
   - dies entspricht der in 5.21 beschriebenen Prozedur -
04 und Ausgabe des entsprechenden Zeichens

ASSEMBLER-Beispiel "44-vorzpruefen" (C64):

```
01 19700 lda #224
   19702 ldy #185    ; Anfangsadresse der ROM-Zahl -0.5
   19704 jsr 48034   ; Zahl nach FAC1 mit MEMFAC
02 19707 jsr 48171   ; SGNFAC prüft Vorzeichen

03 19710 beq 19723   ; 0 im (A)-Register (also Zahl weder positiv
                       noch negativ)? ja ===> Sprung
   19712 rol         ; Linksrotation von <A>
   19713 bcc 19719   ; C-Flag gelöscht (also war 1 in (A))
                       ja ===> zur Ausgabe mit "+" springen
   19715 ldx #45     ; sonst Code für "-" laden und
   19717 bne 19725   ; zum Ausgabeteil springen
   19719 ldx #43     ; Code für "+" laden und
   19721 bne 19725   ; nächsten Befehl überspringen

04 19723 ldx #48     ; Code für "0" laden und
   19725 stx 2023    ; in der unteren rechten Bildschirmecke aus-
                       geben
   19728 rts
```

Aufgaben:
Probieren Sie diese Routine auch mal mit anderen Zahlen aus, die Sie im RAM oder ROM kennen.

Schreiben Sie eine Routine, die die Winkelfunktionen auf ihre Vorzeichen überprüft und den richtigen Winkel im Bereich bis 360 Grad ausgibt.
Dabei dürfen die Funktionswerte nicht willkürlich gewählt werden, weil z.B. für einen bestimmten Winkel Sinus- und Cosinuswerte festliegen.

Arithmetik-ROM-Routinen

Label	C64	40/80XX
ADD0.5	47177=$b849 FAC1 + 0.5 ---> FAC1	51583=$c97f
FMAL10	47842=$bae2 FAC1 mal 10 ---> FAC1	52248=$cc18
FDIV10	47870=$bafe FAC1 / 10 ---> FAC1	52276=$cc34
FMINUS	49076=$bfb4 -FAC1 ---> FAC1	53579=$d14b
FACABS	48216=$bc58 abs(FAC1) ---> FAC1	53622=$cd8e
ADD	47210=$b86a FAC2 + FAC1 ---> FAC1	51616=$c9a0
M-ADD	47207=$b867 MEMORY (A/Y) + FAC1 ---> FAC1	51613=$c9a0
SUB	47187=$b853 FAC2 - FAC1 ---> FAC1	51593=$c989
M-SUB	47184=$b850 MEMORY (A/Y) - FAC1 ---> FAC1	51590=$c986
INTMUL	45900=$b34c <113/114>=<$71/72> mal MEMORY (95/96)=($5f/60) ---> (X/A)	50295=$c477 <110/111>=<$6e/6f> (92/93)=($5c/5d)
MULT	47659=$ba2b FAC2 mal FAC1 ---> FAC1	52065=$cb61
M-MULT	47656=$ba28 MEMORY (A/Y) mal FAC1 ---> FAC1	52062=$cb5e
DIV	47884=$bb0c FAC 2 / FAC 1 ---> FAC 1	52296=$cc48

M-DIV 47887=$bb0f 52293=$cc45
MEMORY (A/Y) / FAC1 ---> FAC1

SQRFAC 49009=$bf71 53512=$d108
Quadratwurzel aus FAC1 ---> FAC1

POTRAD 49019=$bf7b 53522=$d112
FAC2 hoch FAC1 ---> FAC1

M-POT 49016=$bf78 53519=$d10f
MEMORY (A/Y) hoch FAC1 ---> FAC1

LOGNAT 47594=$b9ea 52000=$cb20
ln(FAC1) ---> FAC1

EHOCHF 49133=$bfed 53636=$d184
e hoch FAC1 ---> FAC1

SINUS 57963=$e26b 53897=$d289
sin(FAC1) ---> FAC1 (Bogenmaß RAD)

COSIN 57956=$e256 53890=$d282
cos(FAC1) ---> FAC1 (Bogenmaß RAD)

TANG 58036=$e2b4 53970=$d2d2
tan(FAC1) ---> FAC1 (Bogenmaß RAD)

ARCTAN 58126=$e30e 54060=$d3c2
arctan(FAC1) ---> FAC1 (Bogenmaß RAD)

POLNOM 57433=$e059 53741=$dled
Polynomwert aus Tabelle ab (A/Y) ---> FAC
1. Byte der Tabelle = Polynomgrad n
folgende Bytes enthalten die Koeffizienten a_n
bis a_0 als reelle Zahlen in 5-er Gruppen

CMPFAC 48219=$bc5b 52625=$cd91
vergleicht FAC1 mit MEMORY (A/Y)
FAC < MEM ---> <A>:=255
FAC > MEM ---> <A>:= 1
FAC = MEM ---> <A>:= 0

SGNFAC 48171=$bc2b 52577=$cd61
Vorzeichen von FAC1 ---> (A)
"+" ---> <A>:= 1
"-" ---> <A>:=255
<FAC1> = 0 ---> <A>:=0

MEMFAC 48034=$bba2 52440=$ccd8
reelle Zahl aus MEMORY (A/Y) ---> FAC1

MEMFC2 47756=$ba8c 52162=$cbc2
reelle Zahl aus MEMORY (A/Y) ---> FAC2

FACMEM 48087=$bbd7 52493=$cd0d
FAC1 (reell) ---> MEMORY (X/Y)-Anfangsadresse

FAC1/2 48143=$bc0f 52549=$cd45
<FAC1> ---> FAC2

FAC2/1 48124=$bbfc 52530=$cd32
<FAC2> ---> FAC1

ZUFALL 57495=$e097 53801=$d229
53804=$d22c
Zufallszahl aus 0 bis 1 ---> FAC1,
abhängig von der Zeit bzw. von <A>

FLPOUT 43708=$aabc 53133=$cf8d
Ausgabe von <A>; C64: mit anschließendem CR

FPOUTX 48599=$bdd7 53133=$cf8d
C64: Ausgabe von <FAC> als Betrag, kein CR
C64: mit Vorzeichen: LDY #1 - JSR FLPOUT

BSOUT(CHROUT) 65490=$ffd2 KERNAL 65490=$ffd2
Ausgabe von <A> als ASCII-Code

6

Bildschirmoperationen

6 Bildschirm-Operationen

6.1 Ausgabe einer Integerzahl mit INTOUT

Die ROM-Routine **INTOUT** gibt eine Integerzahl aus dem Bereich -32768 bis +32767 auf den Bildschirm aus. Wir haben sie im vorigen Kapitel schon einmal verwendet. Hier nun die genauere Beschreibung dazu:

Vorbedingung:
Die Integerzahl steht zerlegt in LO- und HI-Byte in den Registern (X) und (A).

ASSEMBLER-Beispiel "45-intout" (C64):

```
- lda #0        ; HI
  ldx #200      ; LO
- jsr 48589     ; Bildschirmausgabe: 200
  rts
```

Bei negativen Ganzzahlen ist das HI-Byte größer/gleich 128. Das achte Bit ist also gesetzt.
(Siehe dazu auch ASSEMBLER-Beispiel bei 4.1.2/Teil 3.)

6.2 Ausgabe einer reellen Zahl mit FLPOUT

Reelle Zahlen können nur über den Floatpointakkumulator FAC1 ausgegeben werden.
Wie diese Zahlen erzeugt werden, wurde ausführlich in Kapitel 4 beschrieben.

Vorbedingung:
Die auszugebende Zahl steht im FAC1.

ASSEMBLER-Beispiele: Siehe Kapitel 4 und 5!

6.3 Ausgabe eines ASCII-Zeichens mit CHROUT(BSOUT)

Jedes der 255 Zeichen aus dem CBM-Zeichensatz kann über das (A)-Register ausgegeben werden. Dazu gehören auch die nicht druckbaren Zeichen von 0 bis 31 und 128 bis 161, also z.B. 'RETURN', 'DEL' als Cursorsteuerzeichen usw.
Die Routine **BSOUT**, oft auch als **CHROUT** bezeichnet (von character out), ist als Ausgabe-Routine recht vielseitig verwendbar, da sie nicht nur für den Bildschirmausdruck, sondern auch für die Ausgabe auf den Drucker, das Floppy oder irgendein anderes Gerät geeignet ist. Wir werden sie deshalb gut im Auge behalten und bei Bedarf wieder anwenden (vor allem in Kapitel 9, das sich mit der Peripherie befaßt).
Da **BSOUT** zu den häufig gebrauchten Einsprüngen gehört, steht es in einer Sprungliste am Ende des ROM-Bereichs zusammen mit weiteren ständig verwendeten Routinen-Anfängen (sog. KERNAL-Routinen). Diese Adressen sind sogar für die verschiedenen CBM-Rechner zumindest teilweise gleich.

Vorbedingung:
Das auszugebende Zeichen muß im (A)-Register stehen.

ASSEMBLER-Beispiel "46-bsout/chrout" (C64 bis 80XX):

```
- lda #43      ; Code für '+'
  jsr 65490    ; CHROUT druckt an momentane Cursorposition: +
- lda #13      ; Code für RETURN
  jsr 65490    ; CHROUT gibt Carriage Return aus
- lda #56      ; Code für '8'
  jsr 65490    ; CHROUT druckt 8 an Zeilenanfang
  rts
```

Ein Vergleich mit BASIC zeigt, daß mit **CHROUT** alle Ausgaben entsprechend dem BASIC-Befehl PRINT CHR$(X); erfolgen, so daß auch Operationen wie die folgenden ausgeführt werden:

Befehl	C64	40/80XX
scroll up	17	25
scroll down	145	153
Zeile löschen		21
Zeile einfügen		149
Zeilenende löschen		22

Zeilenanfang löschen		150
Fenster setzen li/ob		15
dto.	re/unt	143
Textmodus ein	14	14
Graphikmodus ein	142	142

Die im Direktmodus mit den Tasten zu bedienenden Funktionen werden im Programm mit dem entsprechenden ASCII-Code aufgerufen. (Siehe Beispiel!)

6.4 Vorbereitete Zeichenausgaben

Im ROM der CBM-Rechner sind bereits Routinen vorhanden, die den AKKU mit dem entsprechenden Code laden und sofort ausgeben.
Im Anschluß an dieses Kapitel sind die Labels mit den zugehörigen Anfangsadressen aufgelistet.

6.5 Cursorposition festlegen

Die bisher behandelten Routinen erzeugen immer einen Ausdruck von der momentanen Cursorposition aus.
Für eine gesteuerte Bildschirmgestaltung ist es aber notwendig, eine bestimmte Position vorgeben zu können.

6.5.1 Cursorposition C64

Beim C64 läßt sich auf sehr angenehme Weise der Cursor an eine bestimmte Bildschirmstelle aufsetzen, da eine eigene Berechnungsroutine im ROM gegeben ist.

Ablauf:
- Zeilennummer in der Zeropageadresse (214) bereitstellen: **CURZEI**=(214)
- Spaltennummer in **CURSPA**=(211) bereitstellen
- Aufruf der Routine **CURPOS** zur Cursorpositions-Bestimmung

ASSEMBLER-Beispiel "47-cursorposi" (nur C64):

```
- ldy #10      ; Zeile 10 (Beispiel!)
  sty 214      ; nach CURZEI

- ldx #5       ; Spaltennummer 5 (Beispiel!)
  stx 211      ; nach CURSPA

- jsr 58732    ; CURPOS setzt Cursor auf Zeile 10/Spalte 5

- lda #5       ; Integerzahl <HI>=5
  ldx #12      ;             <LO>=12
  jsr 48589    ; mit INTOUT zur Kontrolle ausgeben: 1292
  rts
```

In Abschnitt 10.4 kommen wir noch einmal auf dieses Thema zu sprechen, wenn wir von BASIC aus den Cursor mit Hilfe dieses kleinen Maschinenprogramms setzen.

6.5.2 Cursorposition 40XX/80XX

Bei den 40/80XX-Geräten ist dieses Problem nicht ganz so einfach zu lösen, weil die Cursorsteuerung nicht über eine analoge Berechnungsroutine läuft. Vielmehr orientiert sich der Rechner direkt an den Anfangsadressen der Bildschirmzeilen.
Als Beispiel schauen wir uns dieses Problem für den CBM 8032 an. Das Bildschirm-RAM beginnt bei 32768 = $8000.
Die Anfangsadresse der Zeile 0 (erste Zeile!) hat demnach den Wert LO=0/HI=$80 bzw. LO/HI=0/128, die Zeile 1 LO/HI=80/128 usw.

Diese Anfangsadressen sind nun im ROM bereits abgelegt und zwar getrennt als LO-Bytes der Zeilenanfänge **BSADLO** und in einem weiteren Datenblock als HI-Bytes mit dem Anfang **BSADHI.**

Label	C64	40/80XX
BSADLO	60656=$ecf0	59221=$e755
BSADHI	217=$d9	59246=$e7be

Beide Datenblöcke umfassen je 25 Bytes, weil es eben 25 Bildschirmzeilen gibt.

Gleichzeitig wird in den Zeropage-Adressen (196) und (197) ein Zeiger verwaltet, der immer die Anfangsadresse LO/HI derjenigen Zeile enthält, in der sich der Cursor augenblicklich befindet. Dabei ist es gleichgültig, ob der Cursorfleck blinkt oder unsichtbar bleibt.
Steht der Cursor beispielsweise irgendwo in Zeile 1 (also der zweiten Zeile von oben), dann weist der Zeilenpointer **ZEIPTR** die Bytes 80/128 mit LO/HI auf.

Die Zeile, in der sich der Cursor gerade aufhält, wird auch noch in der Adresse **CURZEI**=(216) gespeichert.

Um auch die Cursorspalte noch zu bestimmen, wird die Zeropage-Adresse (198) mit einem Wert von 0 bis 79 (beim 80-Zeichen-Schirm) belegt. Diese Zeropageadresse nennen wir **CURSPA.**

Wir können also jederzeit
a) die aktuelle Cursorposition Zeile/Spalte aus den Adressen 216 und 198 abfragen,

b) die absolute Bildschirmadresse der Cursorposition aus der Zeilen-Anfangsadresse <196/197> plus Spaltenposition <198> berechnen,

c) den Cursor an jede beliebige Bildschirmstelle schicken. Leider genügt es nicht, einfach die Adressen **CURSPA** und **CURZEI** zu belegen.

Schauen wir uns deshalb das notwendige Verfahren einmal genauer für den 80XX an. Es ist auch für den C64 anwendbar, aber warum sollen wir es dort komplizierter machen, wenn es schon einmal so einfach ging (siehe 6.5.1).

Zielsetzung:
Der Cursor soll in Zeile 12, Spalte 5 aufgesetzt werden. Das entspricht der Bildschirmadresse 33733.

Vorbedingungen:
Die Zeilenanfangsadressen stehen im ROM.
8032: LO ab 59221 / HI ab 59246
C64 : LO ab 60656 / HI ab 217

Ablauf:
01 Spalte 5 nach Spaltenadresse (198)
02 Zeile 12 nach Zeilenadresse (216)
03 HI-Byte der Anfangsadresse der Zeile 12 nach (197)
Es muß also das 12. Byte nach dem Anfang des Zeilenadressen-Blocks HI geholt werden.

04 dto. LO-Byte nach (196)
Wir brauchen dazu das 5. Byte nach der Anfangsadresse des Zeilenanfang-LO-Blocks.

05 Ausgabe der Integerzahl aus (A/X) zur Kontrolle

ASSEMBLER-Beispiel "48-cursorposi" (80XX):

```
01 ldx #5       ; Spalte 5 nach
   stx 198      ; Zeropageadresse (198)=CURSPA
02 ldy #12      ; Zeile 12
   sty 216      ; nach Adresse (216)=CURZEI
03 lda 59246,y ; HI-Byte der Anfangsadresse von Zeile 12 nach (A)
   sta 197      ; und in (197)=ZEIPTR-HI speichern
04 lda 59221,y ; LO-Byte der Anfangsadresse von Zeile 12 nach (A)
   sta 196      ; und in (197)=ZEIPTR-LO speichern
```

<196/197> enthält nun den Wert 33733 in der Form LO/HI.
Nachdem auch die Spaltenposition mit <198> bereits festgelegt ist, kann die Bildschirmausgabe erfolgen:

```
05 jsr 53123  ; INTOUT gibt <A/X> als Integerzahl aus.
   rts
```

Da im (A)-Register noch 12 und im (X)-Register noch 5 steht, erscheint auf dem Bildschirm die Zahl 1292.
Dabei ist noch zu beachten, daß bei allen Zahlenausdrucken, reell oder integer, ein Zeichen mehr für das Vorzeichen benötigt wird. Die erste Ziffer der Zahl 1292 erscheint also in Spalte 6, da die Zahl positiv ist und das Vorzeichen '+' nicht automatisch ausgegeben wird. Stattdessen erscheint eben ein Blank.

6.6 Ausgabe eines Strings mit STROUT

Um eine Zeichenfolge auf den Bildschirm zu drucken, muß diese im ROM oder RAM enthalten sein.

Vorbedingungen:
Die Anfangsadresse der Zeichenfolge muß bekannt sein.
Die Länge des Strings wird benötigt.
Nehmen wir für unsere Demonstration an, wir wollen das Wort 'COMMODORE' ausgeben, das u.a. auch beim Einschalt-Reset auf dem Bildschirm erscheint.
Beim C64 steht es im ROM ab Adresse 58494=$e47e bzw. LO/HI= 126/228. Die Länge dieses Wortes beträgt 9 Zeichen.

Ablauf:
01 Die Anfangsadresse LO/HI des Strings wird nach **STRADR** gebracht, beim C64: (34/35), bei 80XX: (31/32).
02 Die Stringlänge **LEN** kommt ins (X)-Register.
03 Die Routine **STROUT** gibt den String ab aktueller Cursorposition auf dem Bildschirm aus.

ASSEMBLER-Beispiel "49-stringout" (C64):

```
01 lda #126     ; Stringanfangsadresse LO
   sta 34       ; nach STRADR-LO
   lda #228     ; Stinganfangsadresse HI
   sta 35       ; nach STRADR-HI

02 ldx #9       ; Stringlänge LEN nach (X)
03 jsr 43813    ; STROUT gibt 9 Zeichen ab Adresse (58494) aus:
   rts            commodore
```

Probieren Sie nun mal andere Längen **LEN** aus. Wenn Sie z.B. statt 9 die Zahl 12 verwenden, erhalten Sie "commodore 64".

6.7 Umwandlung des FAC-Inhalts in einen String mit FACSTR

Normalerweise besorgt die bereits besprochene Routine **FLPOUT** die Umwandlung einer im FAC1 stehenden Zahl in einen druckbaren String. Das ist deshalb notwendig, da die Zahl im FAC erst deco-

diert und als Dezimalzahl dargestellt werden muß.

Wie sich in den nächsten Abschnitten gleich zeigen wird, ist es aber manchmal sehr nützlich, z.B. eine reelle Zahl zunächst in einen String zu verwandeln, daran Formatierungsoperationen (die der CBM nicht beherrscht) vorzunehmen und dann erst auszugeben.

Die ROM-Routine **FACSTR** wandelt den FAC1-Inhalt in die Ziffernfolge des entsprechenden Zahlenwerts um und legt sie ab Adresse 256... ab. Das letzte Zeichen dieser Folge ist immer ein Byte 0.

Unterschreitet der Zahlenwert im FAC1 den Betrag 0.01, so erfolgt die Umwandlung in Exponentialdarstellung.
Um bei unbekannten Ergebnissen festzustellen, ob dies der Fall ist, untersucht man die Zeropage-Adresse (94). Hier steht nach der Umwandlung in einen String immer der Exponent zur Basis 10. Wurde eine gewöhnliche Dezimalzahl erzeugt, ist <94> = 0.
(Beim 40/80XX ist das die Zeropage-Adresse (91)).

Die Routine **FLPOUT** benützt also als Teilprogramme **FACSTR** und **STROUT.**

Im folgenden Beispiel geben wir die Zahl -32768 auf diese Weise aus.

Vorbedingung:
Die reelle Zahl -76,7041703 steht ab (58106) im ROM. (Ist beim C64 erfüllt.)

Ablauf:
01 Zahl -76,7041703 in den FAC1 bringen.
02 <FAC 1> in einen String umwandeln
03 Stringausgabe vorbereiten
04 Zeichenfolge -76,7 ausgeben

ASSEMBLER-Beispiel "50-flpstring" (C64):

```
01 lda #250      ; LO von Anfangsadresse (58106)
   ldy #226      ; dto. HI
   jsr 48034     ; MEMFAC holt -76,7041703 nach FAC1

02 jsr 48605     ; FLPSTR wandelt FAC1 in String um und legt
                   ihn ab 256 an.
```

```
03 lda #0        ; Stringanfang LO
   sta 34        ; nach STRADR-LO
   lda #1        ; Stringanfang HI (=1mal256)
   sta 35        ; nach STRADR-HI
   ldx #5        ; Stringlänge 5 für fünfstellige Zahl

04 jsr 43813     ; STROUT druckt auf Bildschirm: -76,7
   rts
```

Diese Strings, die aus der Umwandlung des FAC1 kommen, werden immer mit einem 0-Byte abgeschlossen.
Die Routine **STR-0** erkennt dies und gibt automatisch nur die Zeichen bis zum Stringende aus, so daß keine Stringlänge angegeben werden muß. Da sich alle diese Zahlen ab Adresse (256)... befinden, genügt es, die Register (A/Y) mit 0/1 zu belegen. Die Routine erkennt dies als Stringanfang.

Will man die Stringausgabe begrenzen, dann setzt man ein Null-Byte hinter das gewünschte letzte Zeichen.
Das obige Beispiel vereinfacht sich im dritten Teil wie folgt:

```
   ...
03 lda #0        ; LO
   ldy #1        ; HI der Anfangsadresse 256
   sta 262       ; Null als Stringbegrenzer setzen
04 jsr 43806     ; STR-0
   rts
```

6.8 Anwendung: eine PRINT USING - Routine

An dieser Stelle bietet es sich an, ein komplexeres Modul zu diskutieren, das gleichzeitig den Vorteil hat, praktisch anwendbar zu sein und eine Schwäche des CBM-Betriebssystems auszumerzen. Es handelt sich um eine Routine, die Zahlen **formatiert** ausgeben kann.
Es gibt darüber bereits eine ganze Reihe Veröffentlichungen, die aber mehr oder weniger dürftig erläutert sind, so daß die eigentlichen Lernschritte fast immer in einem Wust von Zahlen und Variablen untergehen.
An diesem Modul lassen sich eine Vielzahl von Problemen aufrollen, so daß es sich lohnt, hier etwas länger zu verweilen und die eine oder andere Operation etwas genauer zu beleuchten.

Doch nun ans Werk:
Unsere Routine soll folgenden Forderungen genügen:

a) Die Zahl der Vorkommastellen soll frei gewählt werden können,
b) die Zahl der Nachkommastellen ebenfalls.
c) Man kann zwischen Dezimalpunkt und Dezimalkomma wählen.
d) Das Vorzeichen '+' kann wahlweise mit ausgegeben werden.
e) Das Vorzeichen steht immer direkt vor der Zahl.
f) Die CBM-eigene Verstümmelung der Dezimalbrüche (z.B. '-.5') wird in die übliche Form mit einer Vorkomma-Null gebracht (z.B. '-0.5').

Einschränkung:
Exponentialdarstellungen werden nicht formatiert.

Zusammengefaßt:
Die ausgegebenen Zahlen haben die vorher festgelegte Form:
..###,###.. oder ..##.###.. oder +..###,###.. usw.

Solche Routinen sind z.B. in BASIC-Spracherweiterungen enthalten, die sich aber nur nach dem Cracken des entprechenden Tool-Kits für Maschinen-Programme anzapfen lassen.
Wie man diese Module auch für BASIC-Programme verwenden kann, werden wir später besprechen. Jedenfalls bleiben wir auch hier unserem Konzept treu, das Modul so aufzubauen, daß es frei verschiebbar irgendwo im RAM oder auch im EPROM arbeiten kann.
Das bedingt wieder den Verzicht auf Unterprogrammsprünge mit dem Befehl 'JSR XXXX', wenn die Adresse 'XXXX' im Modulbereich selbst liegt.

Die folgenden Ausführungen verlangen vom Leser sicherlich sehr viel Konzentrationsvermögen, bieten ihm aber dafür auch einen Einblick in ASSEMBLER-Strukturen. Und vielleicht ist der eine oder andere Kniff doch noch unbekannt oder nicht ganz durchschaut gewesen.

Vorbedingungen:
Wir benötigen 6 freie Adressen als Flags, Indices oder Zähler.
In unserem Beispiel werden verwendet:

(900) Zahl der festgestellten Vorkommastellen
(901) Anzahl der gewünschten Vorkommastellen ##,...
(902) Anzahl der gewünschten Nachkommastellen ...,##

(903) Stellung x des Endezeichens (Byte 0) in der Seite 1, also 256,x

(904) Vorzeichenflag
Wird mit Code 43(+) vorbelegt, falls bei positiven Zahlen das Vorzeichen mit ausgegeben werden soll.

(905) Flag für Dezimalmarke
Wird mit Code 44 (,) vorbelegt, falls statt des Dezimalpunkts ein Dezimalkomma ausgegeben werden soll.

Die auszugebende Zahl muß bereits im ROM oder RAM vorhanden und die Anfangsadresse muß bekannt sein.

6.8.1 Ablauf - Struktur:

Das Beispiel wird für die Zahl 2pi durchgezogen, die beim C64 ab (58121)=(009/227) steht.

Das gesamte Programm läßt sich in 11 Teile gliedern.

Teil 1: Initialisierung

- In unserem Beispiel werden das '+'-Flag, das Komma-Flag gesetzt und die Zahl der Vorkommastellen auf 5, die der Nachkommastellen auf 3 festgelegt.
- Die Zahl 6.28318531 wird aus dem ROM geholt und zur Kontrolle ausgedruckt. Dabei bleibt die Ziffernfolge ab (256).. erhalten.
- Es erfolgt ein Carriage Return mit **CHROUT**, um einen direkten Vergleich zwischen dem unformatierten und formatieren Ausdruck sichtbar zu machen.

Flags und Bytes sehen nach Teil 1 also so aus:

```
<900> =   0              <256> =  32  " "
<901> =   4              <257> =  54  "6"
<902> =   2              <258> =  46  "."
<903> =   0              <259> =  50  "2"
<904> =  43  "+"         <260> =  56  "8"
<905> =  44  ","         <261> =  51  "3"
                         <262> =  49  "1"
                         <263> =  56  "8"
                         <264> =  53  "5"
                         <265> =  51  "3"
                         <266> =  49  "1"
                         <267> =   0
                         <268> =   0
```

Teil 2: Dezimalpunkt suchen und Vorkommastellen feststellen

- Die höchste Adresse, in der eine Ziffer der Dezimalzahl stehen kann, ist 266.
  Wir untersuchen daher von dort aus abwärts die Zeichenfolge nach dem Dezimalpunkt.
- Wird er gefunden, erniedrigen wir den Zähler (X) und erhalten die momentane Zahl der Vorkommastellen, die wir nach (900) retten.
- Wird kein Punkt gefunden, dann handelt es sich bei der Zahl um eine ganze Zahl, deren Zeichenfolge mit dem Byte 0 abschließt. In diesem Fall wird Teil 3 übersprungen.

Teil 3: Bei Dezimalzahl Endezeichen durch Ziffer 0 ersetzen

- Das Endezeichen (Byte Null) wird - wieder mit einer Suchschleife von der Basis 257 aus - gesucht und
- durch die Zahl "0" ersetzt (Code 48).
  Anschließend wird Teil 4, die Behandlung der Ganzzahlen übersprungen.

Um eventuellen Verwirrungen zu begegnen:
Das Byte '0', also das Endezeichen einer Ziffernfolge ist der Code für den Klammeraffen.
Die Ziffer '0' hat aber den Code 48.
Wünscht man die Ziffer '0', muß also der Byte-Eintrag 48 sein.

Teil 4: Vorbehandlung einer Ganzzahl

- Durchsuchen der Ziffernfolge nach dem Endezeichen (Byte 0)
- Ersetzen des Bytes 0 durch das Byte 46, also den Code für den Dezimalpunkt.

Das geschieht deswegen, damit man auch eine ganze Zahl (z.B. 10) mit Nachkommastellen ausgeben kann (z.B. 10.000).

Teil 5: Neues Endezeichen setzen

- Zunächst stellen wir fest, ob sich die Zahl als Exponentialzahl ab (256) befindet. Das prüfen wir durch Abfragen der Adresse (94) bzw. 40/80XX:(91). Hier steht ein eventueller Exponent. Sollte dies der Fall sein, gehen wir sofort zum Teil 'Druck'.

Gleichzeitig haben wir damit schon Vorbereitungen getroffen, falls wir später auch exponentielle Darstellungen in Dezimalzahlen verwandeln wollen.

- gewünschte Nachkommastellenzahl laden
- Falls dies 0 sein sollte, muß das Endezeichen direkt hinter

die Ziffer gesetzt werden, also an Stelle des Dezimalpunkts.
- falls eine oder mehrere Nachkommastellen gebraucht werden, addieren wir die bisherigen Vorkommastellen mit den gewünschten Nachkommastellen, erhöhen diese Zahl um 1 wegen der Stelle für den Dezimalpunkt und setzen das Endezeichen mit diesem Index zur Basisadresse (257).
- Da wir später diesen Index als Zähler ab (256) benötigen, erhöhen wir ihn um 1, und weil die später folgende Zählschleife mit Erniedrigen beginnt, erhöhen wir nochmals um 1 und legen damit die Stellung des Endezeichens in (903) ab.

Beim Programmlauf sehen Flags und Bytes nun folgendermaßen aus:

<900> =	1		<256> =	43	"+"
<901> =	4		<257> =	54	"6"
<902> =	2		<258> =	44	","
<903> =	6		<259> =	50	"2"
<904> =	43	"+"	<260> =	56	"8"
<905> =	44	","	<261> =	0	
			<262> =	49	"1"
			<263> =	56	"8"
			<264> =	53	"5"
			<265> =	51	"3"
			<266> =	49	"1"
			<267> =	48	"0"
			<268> =	0	

Teil 6: Art der Vorzeichenausgabe
- Prüfung, ob das Plusflag gesetzt ist
- wenn ja, Vorzeichen der Zahl feststellen
  wenn nein, zu Teil 7 springen
- wenn Vorzeichen positiv, was durch eine Leerstelle in (256) zu erkennen ist, diese durch '+' ersetzen,
  sonst gleich zu Teil 7 springen

Teil 7: Dezimalmarke behandeln
- Prüfung, ob überhaupt Nachkommastellen gewünscht werden.
  Wenn nicht, braucht ein eventueller Dezimalpunkt auch nicht ersetzt werden.
- Sonst Prüfung, ob Dezimalkomma gewünscht wird.
  Wenn ja, Punkt durch Komma ersetzen. Die Zahl der Vorkommastellen haben wir ja in (900) zur Verfügung.

Teil 8: Behandlung der CBM-verstümmelten Zahlen

- Prüfung, ob die Vorkommastellenzahl 0 ist.
  Das ist z.B. bei der CBM-Zahl -.5 der Fall.
- wenn ja, Endestellung nach (Y) und um 1 erniedrigen, da das Vorzeichen bei der folgenden Verschiebung erhalten bleiben soll.
- Alle Zeichen ab (257) um eins nach oben schieben. Damit wird Platz für die Vorkomma-Null geschaffen.
- Null einfügen.
- Da sich die Stellung des Endezeichens und die Zahl der Vorkommastellen jeweils um 1 erhöht haben, müssen diese Indices entsprechend korrigiert werden.

Das Verschieben in der Nähe der Zeropage bedarf noch einer Erläuterung:
Es wäre z.B. möglich, das Vorzeichen nach (255) zu bringen und in (256) eine Null (Code 48) zu schreiben, um damit eine Verschiebung ab (255) zu versuchen. Aber der entsprechende Maschinenbefehl erkennt (255) als Zeropage-Adresse, so daß zum Beispiel für x=10 der ASSEMBLER-Befehl LDA 255,X nicht wie gewünscht den Inhalt von (265) in das (A)-Register holt.
Vielmehr wird das entstehende HI-Byte 1 mißachtet und von 255 aus mit 0 weitergezählt. Geladen wird also wie LDA 9.
Bleibt uns nichts anderes übrig, als den vorgeschlagenen Weg oder eine Abwandlung davon zu programmieren.

Teil 9: Behandlung der Vorkommastellen

- Differenz zwischen vorhandenen und gewünschten Stellen berechnen.
- Bei Gleichheit zu Teil 10 (Ausdruck) springen,
- ebenfalls bei negativen Differenzen, d.h. wenn die Zahl mehr Vorkommastellen besitzt als man vorgegeben hat.
  Das kommt in etwa einer Fehlerbehandlung gleich.
- Sonst Differenz (noch fehlende Stellen) zum Index für Zahlende addieren
- und Index für (Zahlenende +1) als Zähler nach (Y) holen.
- Zeichenfolge nach oben verschieben.

Nachdem sich in unserem Beispiel die Flags nicht geändert haben, sieht die Byte-Folge nun so aus:

```
<900> =    1                    <256> =   43  "+"
<901> =    4                    <257> =   54  "6"
<902> =    2                    <258> =   44  ","
<903> =    6                    <259> =   43  "+"
<904> =   43  "+"               <260> =   54  "6"
<905> =   44  ","               <261> =   44  ","
                                <262> =   50  "2"
                                <263> =   56  "8"
                                <264> =    0
                                <265> =   51  "3"
                                <266> =   49  "1"
                                <267> =   48  "0"
                                <268> =    0
```

Teil 10: Vorkommastellenrest auffüllen

Wie man sieht, sind noch Reste der alten Ziffernfolge im Speicher. Wir füllen sie mit Leerstellen (Code 32) auf.

- Als Zähler können wir den 'Rest' vom (X)-Register gleich weiterverwerten.
- und mit einer Schleife bis hinunter nach (256) alle Adressen mit Leerzeichen belegen:

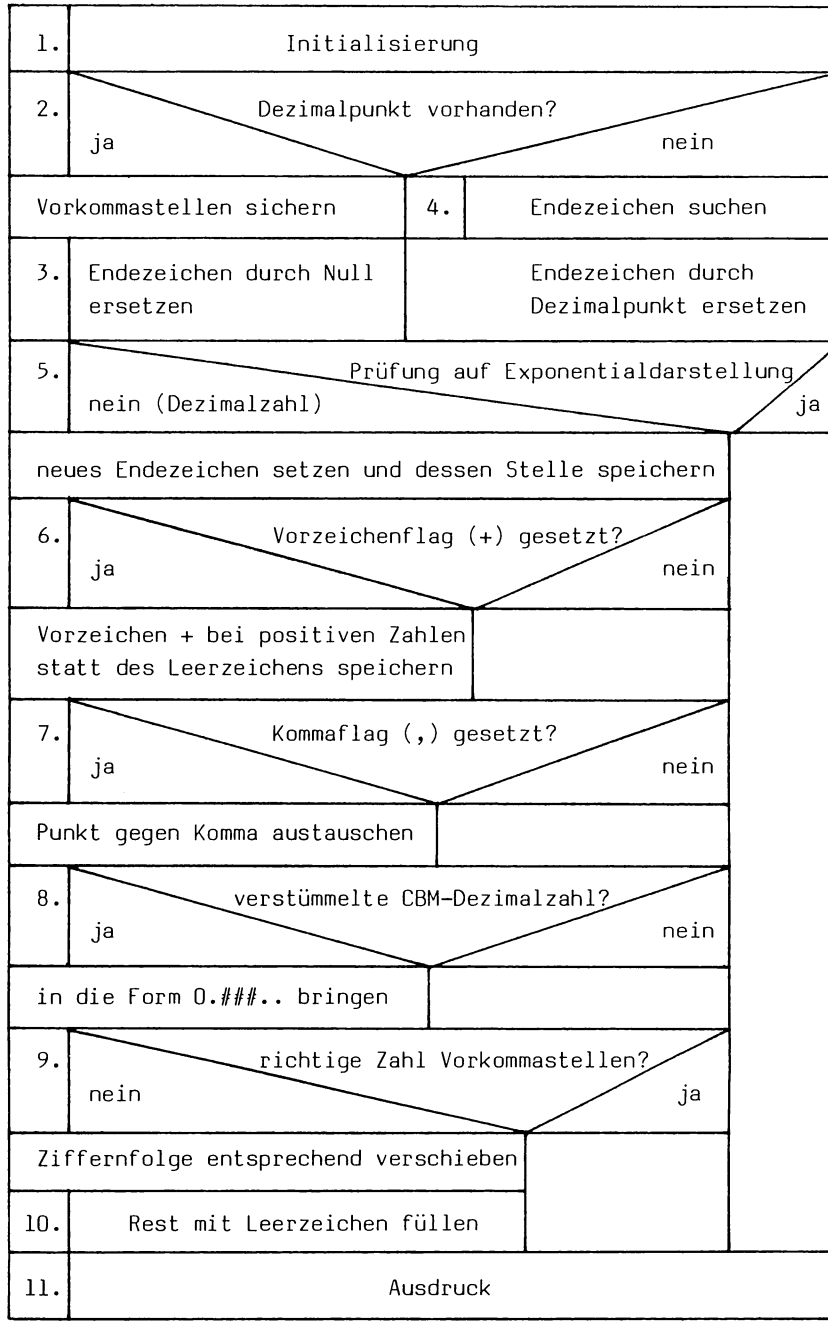
```
                                <256> =   32  " "
                                <257> =   32  " "
                                <258> =   32  " "
                                <259> =   43  "+"
  Flags unverändert             <260> =   54  "6"
                                <261> =   44  ","
                                <262> =   50  "2"
                                <263> =   56  "8"
                                <264> =    0
                                <265> =   51  "3"
                                <266> =   49  "1"
                                <267> =   48  "0"
                                <268> =    0
```

Als Option könnte auch mit Vorkomma-Nullen aufgefüllt werden. Es wäre aber dann nochmal ein Flag, eine Abfrage und eine Versetzung bzw. Belassung des Vorzeichens notwendig.

Teil 11: Ausdruck mit STR-O

- Register (A/Y) mit Anfangsadresse 0/1 belegen
- Ausgabe mit der Routine **STR-O**, da die Ziffernfolge auf jeden Fall mit einem Null-Byte endet.

6.8.2 Struktogramm zur PRINT USING - Routine



6.8.3 ASSEMBLER-Modul "5l-print using" (C64, 40/80XX in Klammern):

Teil 1: Initialisierung

```
- 18125 lda #43      ; '+'-Zeichen setzen
  18127 sta 904
  18130 lda #5       ; Vorkommastellenzahl setzen, also
  18132 sta 901      ; die Form +#####,.... wählen
  18135 lda #3       ; Nachkommastellenzahl wählen, damit
  18137 sta 902      ; endgültige Form:  +#####,## festlegen
  18140 lda #44      ; Komma-Code setzen
  18142 sta 905
  18145 lda #9       ; Anfangsadresse LO von 2pi (Beispiel)
  18147 ldy #227     ; dto. HI
  18149 jsr 48034    ; MEMFAC holt 2pi nach FAC1
 (18149 jsr 52440)
  18152 jsr 43708    ; FLPOUT druckt: 6.28318531
 (18152 jsr 53133)
        nop
-(18155 lda #13)     ; RETURN ausgeben mit
 (18157 jsr 65490)   ; CHROUT, also in nächste Zeile gehen
        nop
```

Teil 2: Dezimalpunkt suchen

```
- 18161 ldx #11      ; Zähler auf 11
  18163 dex          ; Zähler -1
  18164 beq 18194    ; Sprung, wenn Zähler bei 0 angelangt ist
  18166 lda 256,x    ; Ziffernzeichen holen und
  18169 cmp #46      ; auf Punkt überprüfen
  18171 bne 18163    ; weitersuchen, wenn kein Punkt gefunden wird

- 18173 dex          ; Punkt gefunden ---> Zähler erniedrigen
  18174 stx 900      ; jetzt ist in (X) die Anzahl der Vorkomma-
                       stellen, die nach (900) gerettet wird
```

Teil 3: Null bei Dezimalzahl setzen

```
- 18177 ldx #11      ; Zähler wieder auf 11 setzen
  18179 dex          ; Zähler minus 1
  18180 lda 257,x    ; Zeichen (von 267 abwärts) laden
  18183 bne 18179    ; weiter abwärts zählen, bis im (A) eine Null
                       steht (dann wird das Z-Flag gelöscht)
  18185 lda #48      ; Code für Zahl 0 laden,
  18187 sta 257,x    ; dort speichern, wo die Null war
  18190 bne 18213    ; und den Teil 3 überspringen
        nop
```

Teil 4: Behandlung einer eventuellen Ganzzahl

```
- 18194 ldx #10       ; Zähler auf 10
  18196 dex           ; Zähler minus 1
  18197 beq 18204     ; Zähler auf 0 ===> Sprung
  18199 lda 257,x     ; Zeichen laden und auf Code 0 (Endezeichen)
  18202 bne 18196     ; durchsuchen

- 18204 lda #46       ; 0 gefunden ---> Punktcode laden
  18206 sta 257,x     ; und Code 0 durch Code 46 ersetzen
  18209 stx 900       ; Zähler (X) enthält gleichzeitig die Anzahl
                        der Vorkommastellen ---> retten nach (900)
```

Teil 5: Endezeichen neu setzen in Abhängigkeit von der Zahl der geforderten (drei) Nachkommastellen

```
- 18213 lda 94        ; Prüfung auf eventuelle Exp.-Darstellung
 (18213 lda 91)        durch Untersuchen des Exponenten
  18215 bne 18322     ; falls Exponent ungleich Null ===> Drucken

- 18217 lda 902       ; gewünschte Anzahl Nachkommastellen laden
  18220 bne 18228     ; mehr als 0? ===> weiter

- 18222 ldx 900       ; gleich 0 ---> Vorkommastellenzahl laden
  18225 clc           ; und unbedingt die Behandlung
  18226 bcc 18234     ; der Nachstellen überspringen

- 18228 clc
  18229 adc 900       ; Nachkommastellen + Vorkommastellen
  18232 tax           ; Summe nach (X)
  18233 inx           ; und um 1 erhöhen wegen des Dezimalzeichens
  18234 lda #0        ; Endezeichen
  18236 sta 257,x     ; neu setzen
    nop

- 18240 inx           ; Zähler wird später ab (256) benützt und
  18241 inx           ; die Schleife beginnt mit DEX
  18242 stx 903       ; Länge der Ziffernfolge +1 steht jetzt in
                        (X) und wird nach (903) gerettet
```

Teil 6: Vorzeichen festlegen

```
- 18245 lda 904
  18248 cmp #43     ; Vorzeichen '+' vorgeschrieben?
  18250 bne 18262   ; nein ===> weiter

- 18252 ldx 256     ; ja  --->  Vorzeichen  der  Zahl  holen
  18255 cpx #32     ; positiv?
  18257 bne 18262   ; nein ===> weiter
- 18259 sta 256     ; ja  --->  Vorzeichen  '+'  speichern
```

Teil 7: Dezimalmarke setzen

```
- 18262 lda 902
  18265 beq 18280   ; keine Nachkommastellen gewünscht ===>weiter

- 18267 lda 905
  18270 cmp #44     ; Komma gewünscht?
  18272 bne 18280   ; nein ===> weiter

- 18274 ldx 900     ; Anzahl  Vorkommastellen
  18277 sta 257,x   ; Komma statt Punkt setzten
```

Teil 8: Verstümmelung bereinigen

```
- 18280 ldx 900     ; Vorkommastellen  laden
  18283 bne 18311   ; welche vorhanden ===> weiter

- 18285 ldy 903     ; Stellung  des  Endezeichens
  18288 dey         ; als Zähler herrichten
- 18289 dey         ; Schleifenbeginn  zur
  18290 lda 257,y   ; Verschiebung der Zahl um 1
  18293 sta 258,y   ; nach oben
  18296 tya         ; Schleifenende erreicht?
  18297 bne 18289   ; nein ===> weiter verschieben

- 18299 lda #48     ; Code  für  0
  18301 sta 257     ; hinter das Vorzeichen setzen

- 18304 inc 900     ; Vorkommastellenzahl  jetzt  +1
  18307 inc 903     ; Endezeichen jetzt +1
        nop
```

Teil 9/1: Vorkommastellen überprüfen

```
  18314 sec          ; Subtraktion vorbereiten
  18315 sbc 900      ; gewünschte minus vorhandene VK-Stellen

- 18318 beq 18322    ; keine Differenz ===> sofort Ausdruck
  18320 bcs 18330    ; Differenz festgestellt ===> Ausdruck über-
                       springen und zunächst Teil 9/2 bearbeiten
```

Teil 11: Ausdruck

```
- 18322 lda #0       ; Zeiger (A/Y) auf
  18324 ldy #1       ; (256) stellen

- 18326 jsr 43806    ; STR-0 gibt aus: 6,283
 (18326 jsr 47901)
        rts          ; und Ende
```

Teil 9/2: Vorkommastellen einrichten

```
- 18330 clc          ; Addition vorbereiten
  18331 adc 903      ; VK-Differenz plus bisherige Länge
  18334 tax          ; als Zähler nach (X)

- 18335 ldy 903      ; bisherige Länge als 2.Zähler nach (Y)

- 18338 dey          ; Schleifenbeginn
  18339 dex          ; für
  18340 lda 256,y    ; die Verschiebung
  18343 sta 256,x    ; nach oben
  18346 tya          ; Ende erreicht?
  18347 bne 18338    ; nein ---> weiter verschieben
```

Teil 10: Füllzeichen

```
- 18349 dex          ; sonst (X) als Zähler weiterverwenden
  18350 lda #32      ; und mit Leerzeichen
  18352 sta 256,x    ; nicht belegte VK-Stellen auffüllen
  18355 txa          ; Endeprüfung
  18356 bne 18349    ; x>0 ===> weiter füllen
  18358 beq 18322    ; x=0, also fertig ===> zum Ausdruck
```

Die hier im ASSEMBLER-Programm noch eingebauten NOPs lassen sich zum Austesten des Moduls gegen RTS-Befehle umtauschen oder als BREAK-Points verwenden.

Aufgabe:
Erweitern Sie die PRINT USING-Routine, so daß auch Zahlen in Exponentialdarstellung als Dezimalzahlen formatiert ausgedruckt werden.

Aufgabe:
Wandeln Sie die PRINT USING-Routine so ab, daß mit der Formatierung auch eine Rundung erfolgt.
Vorschlag:
- Nachkommastellenzahl zunächst um 1 vergrößern
- Durchlauf der PU-Routine
- vor dem Ausdruck die letzte Stelle untersuchen
- letzte Stelle größer/gleich 5 ---> vorletzte erhöhen
  (Achtung, wenn vorletzte gleich 9 ?!)
- letzte Stelle abschneiden und zum Ausdruck übergehen

6.9 Ausgabe von Hex-Zahlen (nur 40/80XX)

Aus dem MONITOR der CBM-Rechner (nicht C64) lassen sich noch ein paar brauchbare Ausgaberoutinen stibitzen:

6.9.1 Byte in Hex-Form ausgeben mit BYTOUT

Die auszugebende Zahl muß im (A)-Register bereitgestellt werden. Nach Aufruf der Routine **BYTOUT** erscheint die Zahl als zweistellige Hexzahl auf dem Schirm.

Beispiel nur 40/80XX:
Die Zahl 160 soll in Hexform gebracht werden.

ASSEMBLER-Beispiel "52-byte out" (80XX):

```
- lda #160        ; 160 nach (A)
- jsr 55074       ; BYTOUT wandelt 160 um in $a0
```

6.9.2 Vierstellige Hexzahl (Adresse) ausgeben mit ADROUT

Die Zahl muß zerlegt in LO/HI in (251/252) vorhanden sein.

Beispiel nur 40/80XX:
Der Zeiger in (144/145) soll als Hexzahl ausgegeben werden.

ASSEMBLER-Beispiel "53-adressout" (80XX):

```
- lda 144
  sta 251        ; LO übertragen
  lda 145
  sta 252        ; HI übertragen
- jsr 55063      ; ADROUT gibt aus z.B.: $e455
                   Das ist übrigens der Standard-Einsprung für
                   die Interrupt-Routine (IRQ).
```

Siehe zu diesem Thema auch Abschnitt 10.9!

6.9.3 Zwei Zeichen ausgeben mit OUT2

Zwei Zeichen - druckfähig oder auch nicht - können mit dem Einsprung nach **OUT2** hintereinander ausgegeben werden.
Dazu muß das erste Zeichen - genauer gesagt, der Code des ersten Zeichens - nach (X) und der des zweiten nach (A) gebracht werden.
Beide Zeichen werden entsprechend dem ASCII-Code ausgegeben.

Beispiel:
Nach Bildschirm CLR (Code 147) wird der Buchstabe 'X' gedruckt.

ASSEMBLER-Beispiel "54-zwei ausgaben" (80XX):

```
- ldx #147       ; Code für Bildschirm clear
  lda #88        ; Code für 'X'
- jsr 55089      ; OUT2 löscht Bildschirm und druckt:X
```

6.10 Bewegungssimulation - eine Kompaßanzeige

Erinnern Sie sich noch an die Programme "01-pinsel" und "03-taktmodul" aus Kapitel 2?
Dort haben wir ein Anzeigegerät für Richtungsänderungen gebaut.
Mit Hilfe dieses sog. Wendezeigers läßt sich nun die Kompaßrichtung bestimmen, wenn man einen Bewegungsvorgang simulieren will.

Man braucht solche Module immer dann, wenn Orientierungs- oder Navigationsprobleme bei Fahrzeugen (Flugzeugen, Schiffen usw.) zu lösen sind.

Wir sind schon so weit gekommen, daß wir mit dem Programm "taktmodul" eine Bewegung digitalisieren können. In unserem Beispiel haben wir eine Taktfrequenz von $4s^{-1}$ (alle 0.25s ein Takt) gewählt. Diesen Wert wollen wir nun den folgenden Überlegungen zu Grunde legen. Jetzt sind wir nämlich mit Hilfe unserer Arithmetik- und Ausgaberoutinen in der Lage, eine Bewegung um eine Achse zu simulieren und die aktuelle Richtung jederzeit zu bestimmen und als Kompaßanzeige auszugeben.

Unser Ziel:

- Mit dem Joystick (oder der Tastatur) geben wir die Richtung vor, in die wir uns drehen wollen, aber auch die Geschwindigkeit der Drehbewegung.
  Nehmen wir einen Wert aus der Fliegerei an, dann entspricht der Ausschlag von einer Pinselbreite des Wendezeigers einer Drehgeschwindigkeit von 6 Grad pro Sekunde.
  Eine volle Umdrehung (ein Vollkreis) dauert dann genau eine Minute, eine halbe Umdrehung 30 Sekunden usw.
  Anders ausgedrückt: In jeder Sekunde soll sich die Richtung um plus 6 Grad - also nach rechts - verändern, wenn der Pinsel um eine Breite nach rechts auf dem Anzeigegerät steht. Steht er z.B. zwei volle Breiten links, dann ändert sich die Richtung in jeder Sekunde um minus 12 Grad.

- Die Richtung wird in Form einer dreistelligen Zahl als Kompaßkurs dargestellt:
  Dabei gilt: 360 = 000 = Nord
  090 = Ost
  180 = Süd
  270 = West

Nun, können Sie schon vermuten, wie wir die Richtung berechnen?
- Zunächst nehmen wir als Ausgangszustand an, daß beim Programmstart die Richtung Nord (=000) gegeben ist.

- Die Stellung des Wendezeigers finden wir in der Zeropage-Adresse (189), wobei 18 die Mittelstellung, 4 voller Linksausschlag und 32 voller Rechtsausschlag bedeutet.

- In vier Schritten bewegt sich der Pinsel um eine Breite nach links oder rechts, wenn Sie das Programm "01-pinsel" nicht abgewandelt haben. Wir behalten diesen Wert bei (weil er auch bei den 40/80XX-Geräten brauchbar ist).
  Steht z.B. in (180) der Wert 14, dann bedeutet dies, daß eine Linksdrehung mit genau einer Pinselbreite simuliert wird. Die Kompaßanzeige muß also pro Sekunde um 6 Grad reduziert werden.

- Um links und rechts gleich von vornherein mit dem richtigen Vorzeichen zu versehen, subtrahieren wir von der momentanen Anzeige einfach den Mittelwert 18. Dazu halten wir die Zahl -18 als reelle Konstante **K1** im RAM bereit.

Damit ist "geradeaus" = 0
"1 Pinselbreite rechts" = 4
"1 Pinselbreite links" = -4
"2 Pinselbreiten links" = -8
usw.

Für diesen Wert legen wir eine reelle Zahl an, deren Anfang wir mit **STELLUNG** bezeichnen.

<u>Jetzt kann die eigentliche Rechnerei losgehen:</u>

00 Initialisierungen:
Konstante **K1** = -18 ab (17800=$4588)=136/69
Konstante **K2** = 0.375 ab (17805=$458d)=141/69
Konstante **K3** = 360 ab (17810=$4592)=146/69
Konstante **K4** = -360 ab (17815=$4597)=151/69
RICHTUNG = 0 ab (17820=$459c)=156/69
STELLUNG18 = 18 in (180)=($b4) aus "01-pinsel"
Routine zum Cursoreinstellen auf Zeile 10, Spalte 30.

01 Subtrahieren wir von **STELLUNG18**, die vom Wendezeiger geliefert wird, den Wert 18, dann erhalten wir die **STELLUNG** des Wendezeigers in der Form, daß 0 "Mitte", negativer Wert "links" und ein positiver Wert "rechts" bedeutet.
Bei unserer Einteilung bedeutet jeweils 1 Einheit in **STELLUNG** eine Richtungsänderung von 1,5 Grad pro Sekunde, da der Pinsel seine Richtung in Viertelbreiten ändert.

02 Bei einem Takt von 0.25 Sekunden ändert sich die Richtung also um DR=0,25 mal 1,5 mal <STELLUNG>.
(DR bedeutet hier "Differenz der Richtung" pro Takt.)

Fassen wir die beiden ersten konstanten Faktoren zusammen, dann erhalten wir:

DR = 0,375 mal <STELLUNG>

Die reelle Zahl 0,375 halten wir als Konstante **K2** in Form einer reellen Zahl bereit. Sie wird zu Beginn des Programms im Initialisierungsteil erzeugt werden.

03 Die neue Kompaßrichtung **RICHTUNG** ergibt sich jetzt ganz einfach aus der alten Richtung **RICHTUNG** plus der Richtungsänderung **DR**:

<RICHTUNG> + DR ----> <RICHTUNG>

04 Bevor wir nun unsere gefundene Richtung auf dem Bildschirm ausgeben, muß die berechnete Zahl RICHTUNG eventuell erst bereinigt werden, falls sie nicht im Bereich zwischen 0 und 360 liegt.
Nehmen wir an, die alte Richtung war 0 Grad und die Drehung erfolgte nach links, dann liegt jetzt eine negative Zahl vor. Andererseits wird beim Rechtsdrehen schnell der Wert 360 überschritten.

Für diese Randbedingungen legen wir nun fest:
falls <RICHTUNG> < 0 , dann RICHTUNG = RICHTUNG + 360
falls <RICHTUNG> >360 , dann RICHTUNG = RICHTUNG - 360

Wir benötigen zum Rechnen demnach eine zweite Konstante mit dem Wert 360. Nennen wir sie **K3**. Da sie nicht im ROM steht, muß auch sie erst als reelle Zahl erzeugt werden.
Zweckmäßigerweise legen wir auch gleich eine Konstante **K4**= -360 an, denn die Addition läßt sich leichter durchführen als die Subtraktion.

05 Die bereinigte Zahl RICHTUNG legen wir dort ab, wo vorher die alte Richtung stand, denn schon im nächsten Takt wird die neue zur alten Richtung. Auch hier benötigen wir 5 Bytes mit der Anfangsadresse **RICHTUNG**.

06 Die Bildschirmausgabe der Richtung setzen wir neben den Wende-
. zeiger.
. Dazu wandeln wir den Inhalt von <RICHTUNG> in einen String
. um und geben ihn mit **STR-0** aus, nachdem wir eine Null als
07 Endezeichen so gesetzt haben, daß nur eine Integerzahl zum

Ausdruck kommt.
Damit nicht eventuelle Reste des vorhergehenden Kurses zu Falschanzeigen führen, löschen wir vorher aber grundsätzlich die benötigten drei Bildschirmadressen.
Wen das damit verbundene eventuelle Flimmern stört, das nun bei konstanter Richtung auftritt, der muß dafür sorgen, daß in jedem Takt alte und neue Richtung verglichen werden und bei Gleichheit eben gar kein Ausdruck erfolgt.

08 Sie werden mit der Ausgabe auf dem Bildschirm wahrscheinlich noch nicht ganz zufrieden sein, weil der Druckanfang, den wir vor jedem Ausdruck festlegen müssen, immer der gleiche ist, egal ob es sich um eine ein- oder dreistellige Zahl handelt. Doch wie man den Cursor positioniert oder wie man eventuell die PRINT USING-Routine einbaut, das überlassen wir Ihnen. Wir haben das bereits besprochen. Außerdem ist eine Formatierung durch den Ablauf von Teil 07 schon sehr weit vorbereitet.

ASSEMBLER-Programme "55-kompin" und "56-komprechnen" (C64):

00 Initialisierungen "55-kompin"

```
-  17500 ldy #18
   17502 lda #0
   17504 jsr INTFLP   ; 18 in reelle Zahl
   17507 ldx #136
   17509 ldy #69
   17511 jsr FACMEM   ; 18 ab (17800...17804) ablegen
   17514 lda 17801    ; erstes Mantissenbyte holen
   17517 ora #128     ; und Negativbit 7 setzen
   17519 sta 17801    ; Zahl -18 fertig
         nop
-  17523 ldy #119
   17525 lda #1
   17527 jsr INTFLP   ; 375 in reelle Zahl
   17530 jsr FDIV10   ; durch 10 = 37.5
   17533 jsr FDIV10   ; durch 10 = 3.75
   17536 jsr FDIV10   ; durch 10 = 0.375
   17539 ldx #141
   17541 ldy #69
   17543 jsr FACMEM   ; 0.375 ab (17805...17809) ablegen
         nop
-  17547 ldy #104
   17549 lda #1
   17551 jsr INTFLP   ; 360 in reelle Zahl
```

```
   17554 ldx #146
   17556 ldy #69
   17558 jsr FACMEM    ; 360 ab (17810...17814) ablegen
         nop
-  17562 lda #104
   17564 ldy #1
   17566 jsr INTFLP    ; 360 nach FAC
   17569 lda #128
   17571 sta 102       ; Negativbit im FAC setzen
   17573 ldx #151
   17575 ldy #69
   17577 jsr FACMEM    ; -360 nach (17815...17819)
         nop
-  17581 ldx #4        ; Zähler für 5 Einträge
   17583 lda #0
   17585 sta 17820,x    ; reelle Zahl 0 nach (17820...
   17588 dex           ; ...17824)
   17589 bpl 17585
         nop
-  17592 lda #30
   17594 sta CURSPA    ; Spalte 30 wählen
   17596 lda #10
   17598 sta CURZEI    ; Zeile 10 wählen
   17600 jsr CURPOS    ; Cursor positionieren
-  17603 rts           ; fertig
```

Für den 40/80XX ist nur der letzte Teil abzuwandeln (siehe 6.5).

Hauptteil "56-komprechnen"

01 Links/Rechts-Stellung berechnen

```
-  17120 ldy 189       ;
   17122 lda #0        ; Stellung des Pinsels
   17124 jsr INTFLP    ; in reelle Zahl umwandeln
   17127 lda #136
   17129 ldy #69
   17131 jsr M-ADD     ; und -18 addieren (Mitte jetzt 0)
         nop
```

02 Stellung mit Faktor K2=0.375 multiplizieren

```
   17135 lda #141
   17137 ldy #69
   17139 jsr M-MULT    ; mal 0.375
         nop
```

```
03 zur RICHTUNG addieren
   17143 lda #156
   17145 ldy #69
   17147 jsr M-ADD     ; + alte Richtung
         nop
04 Bereich 0 bis 360 für die RICHTUNG sicherstellen
   17151 lda 102       ; Vorzeichen des FAC laden
   17153 rol           ; Byte 7 in C-Flag schieben
   17154 bcc 17167     ; positive Zahl ---> Addition überspringen
   17156 lda #146
   17158 ldy #69
   17160 jsr M-ADD     ; sonst 360 addieren
   17163 clc
   17164 bcc 17188     ; und 2. Überprüfung überspringen
         nop
-  17167 lda #146
   17169 ldy #69
   17171 jsr CMPFAC    ; FAC mit 360 vergleichen
   17174 lsr           ; Bit 0 in C-Flag schieben
   17175 bne 17188     ; nicht größer 360 ===> weiter zu 05
   17177 nop
   17178 lda #151
   17180 ldy #69
   17182 jsr M-ADD     ; zur RICHTUNG -360 addieren
         nop
05 RICHTUNG abspeichern
   17188 ldx #156
   17190 ldy #69
   17192 jsr FACMEM    ; RICHTUNG nach (17820...17824)
         nop
 06 Ausdruck der Richtung als Ganzzahl
-  17196 jsr 17591     ; Cursorposition
   17199 ldx #4        ; Zähler auf 4
   17201 lda #32       ; Leerzeichen laden
   17203 jsr CHROUT    ; und ausgeben
   17206 dex
   17207 bne 17203
   17209 jsr 17591     ; nochmal Cursorposition einstellen
         nop
-  17213 lda #156
   17215 ldy #69
   17217 jsr MEMFAC    ; RICHTUNG wieder nach FAC holen
   17220 jsr FLPSTR    ; und in String wandeln (ab 256...)
```

```
-  17223 ldx #255      ; Zähler initialisieren
   17225 inx           ; Schleifenbeginn für
   17227 lda 256,x     ; Suche nach
   17229 beq 17240     ; Endezeichen 0 ===> Ausdruck 07
   17231 cmp #46       ; oder Dezimalpunkt
   17233 bne 17225     ; nicht gefunden ===> Schleifenanfang
-  17235 lda #0        ; gefunden ---> Endezeichen 0 setzen
   17237 sta 256,x
```

07 Bildschirmausgabe

```
   17240 jmp STR-0     ; Ausgabe ab (256) bis Zeichen 0
```

Anmerkung zu 07: (Y) steht noch durch **FLPSTR** auf 1. Daher braucht der Stringanfang (Y/A) nicht extra belegt werden, zumal auch (A) den Wert 0 enthält.

Speichern Sie nun diese Teile insgesamt von (17120) bis (17245) ab. Wir können hier gleich die beiden Einsprünge "55-kompin" bei (17500) und "56-komprechnen" bei (17120) festhalten.

Und jetzt geht es im Baukastensystem mit den Teilen aus Kapitel 2 weiter:
Schließen Sie Programmteile "01-pinsel" und "02-vorpinsel" jeweils mit einem RTS ab, entfernen Sie aber vorher die ursprünglichen Verzweigungen.

Das komplette Kompaßprogramm beginnt nun mit dem Setzen des IRQ-Vektors auf (17000), wo die Routine "04-taktmodul" steht, die wir ab (17000) bis (17040) unverändert übernehmen.
Als nächstes werden die Initialisierungen für "01-pinsel" und für "56-komprechnen" aufgerufen.
Nun beginnt die eigentliche Hauptschleife mit einer Abwandlung seres Programms "05-takttest":

ASSEMBLER-Programm "57-taktflag" (C64):

```
-  17100 lda 1010     ; Taktflag
   17103 beq 17113    ; nicht gesetzt ===> Ende
-  17105 lda #0       ; gesetzt ---> zurücksetzen
   17107 sta 1010
-  17110 jsr 17120    ; aber "100-komprechnen" anspringen
   17113 rts          ; Rücksprung bei Takftlag = 0
```

Nach diesem Unterprogramm folgt als nächstes der "01-pinsel", der

eventuelle Richtungsänderungen aufnimmt.

Den Schluß der Hauptschleife bildet eine kleine Abbruch-Routine, damit wir das Programm wieder verlassen können. Sie fragt das Tastatur-Byte **KEY** ab und bricht bei der Leertaste ab. Damit andere Programme wieder vernünftig laufen, stellen wir den IRQ-Vektor wieder standardmäßig auf 59953=$ea31 also 49/234=LO/HI.

ASSEMBLER-Programm "58-abbruch" (C64):

```
- 18100 lda #203      ; KEY
  18102 cmp #60       ; Leertaste?
  18104 beq 10109     ; ja ===> Abbruch
  18106 jmp 24518     ; nein ===> Anfang Hauptschleife
  18109 sei
  18110 lda #49
  18112 sta 788       ; IRQ-LO
  18115 lda #234
  18117 sta 789       ; IRQ-HI
  18118 cli
  18119 rts           ; Ende
```

Hier noch einmal alles auf einen Blick für den "kompaß":

ASSEMBLER-Programm "59-kompaß":

```
01 IRQ-Vektor auf (17000) einstellen:
   24500 sei
   24501 lda #104
   24503 sta 788
   24506 lda #66
   24508 sta 789
   24511 cli
02 Initialisierungen - Vorbereitungen
   24512 jsr 16000    ; Unterprogramm "02-vorpinsel"
   24515 jsr 17500    ; Unterprogramm "55-kompin"
03 Hauptschleife:
   24518 jsr 17100    ; "57-taktflag" und "56-komprechnen"
   24521 jsr 18000    ; "01-pinsel"
   24524 jmp 18100    ; "58-abbruch"
```

Gestartet wird "kompaß" mit SYS 24500.

Wir haben die Programmteile durch große Lücken gut getrennt. Schieben Sie doch mal alles zur Übung auf engsten Raum zusammen!

Übrigens: Wenn alles funktioniert, haben Sie bereits das Herzstück eines Simulatorprogrammes für jede Art von Drehbewegungen in Händen. Von hier aus können Sie beliebig weiterarbeiten:

- Nehmen Sie eine bestimmte Geschwindigkeit an, mit der Sie sich bewegen - oder noch besser: geben Sie diese über den Joystick ein. Vordrücken macht schneller, zurückziehen langsamer.
- Sie können nun mit Hilfe von x/y-Koordinaten jederzeit den aktuellen Standort zu einem beliebigen Fixpunkt aus angeben. Dazu müßten Sie aber etwas Vektorrechnung beherrschen.
- Auch das Bewegen im dreidimensionalen Raum kann man damit hervorragend simulieren.

Wir wollen aber hier den Rahmen mit Spezialitäten nicht sprengen. Tüfteln Sie ruhig selbst weiter. Es macht unheimlich Spaß - wenn alles irgendwann einmal so klappt wie vorgesehen.

Bildschirm-Ausgabe-Routinen

Label	C64	40/80XX
CHROUT(BSOUT)	65490=$ffd2 KERNAL	65490=$ffd2
	<A> auf den Bildschirm als ASCII-Zeichen	
INTOUT	48589=$bdcd	53123=$cf83
	Die Integerzahl wird direkt aus (A/X) mit HI/LO auf den Bildschirm gebracht.	
CURPOS	58732=$e5bc	---
	berechnet mit CURZEI und CURSPA Cursorposition	
FPOUTX	48599=$bdd7	53133=$cf8d
	C64: LDY #1 / JSR 48599	
FLPOUT	43708=$aabc	53133=$cf8d
	C64: Ausgabe <FAC> mit anschließendem CR	

FLPSTR	48605=$bddd	53139=$cf93
	<FAC> ---> String ab (256)	
STROUT	43813=$ab25	47908=$bb24
	<X>=Stringlänge	
mit STRADR	<34/35> Stringanfang	<31/32>
STR-0	43806=$able	47901=$bbld
	<A/Y>=Stringadresse; Ende des Strings: Byte 0	
BYTOUT	---	55074=$d722
	<A> ---> Hex ---> Schirm	
ADROUT	---	55063=$d717
	<251/252> ---> Hex ---> Schirm	
OUT2	---	55089=$d731
	<X> ---> Schirm, <A> ---> Schirm	

Adressen

CURZEI	214=$d6	216=$d8
CURSPA	211=$d3	198=$c6
ZEIPTR	209/210=$d1/d2	196/197=$c4/c5

7

Eingabe-ROM-Routinen

7 Eingabe-Routinen

Eingaben werden dem Computer von außen mitgeteilt. Das bedeutet, daß irgendwelche Geräte irgendwelche Signale schicken, die in Byteform - also 8-bitweise - aufgenommen werden.
An dieser Stelle wollen wir nur die Standard-Eingaben besprechen. Das sind solche, die der Rechner über die Tastatur empfangen kann. Wie man mit dem Joystick oder ausgewählten Tasten eine Eingabe auswerten kann, haben wir auf einfache Weise im Kapitel 2 kennengelernt. Hier geht es nun um die ROM-Routinen, mit denen man bequem Daten über die gesamte Tastatur einlesen kann.

7.1 Eingabe eines Zeichens über die Tastatur

Um ein Zeichen nach einem Tastendruck in das (A)-Register zu übernehmen, steht die ROM-Routine **GETIN** zur Verfügung, die komfortablerweise das Zeichen nicht direkt von der Tastatur holt, sondern den Tastatur-Puffer abbaut.
Wurde kein Zeichen gefunden, so enthält das (A)-Register eine 0. Um zu verhindern, daß das Programm einfach weiterläuft, wenn kein Byte über die Tastatur empfangen wurde, müssen wir mit einer kurzen Schleife wieder zur **GETIN**-Routine verzweigen.
Erfolgte aber ein "INPUT", dann wird er im (A)-Register zur weiteren Behandlung bereitgestellt. Bei unserem Beispielprogramm geben wir das Zeichen im Bildschirmcode und mit **CHROUT** aus.

Beispiel:
Eine beliebige Zahl von Tastaturzeichen sollen nach (A) geholt und ausgegeben werden.

Ablauf:
- Mehrfacher Aufruf von **GETIN**
- mit anschließendem Ausdruck
  Wir geben dabei <A> zunächst mit STA XXXX einfach in den Bildschirmspeicher aus.
  Anschließend verwenden wir unsere Ausgabe **CHROUT**.

ASSEMBLER-Beispiel "60-getin" (C64):

```
- 18043 jsr 65508    ; GETIN, ein Zeichen aus Tastatur nach (A)
  18046 beq 18043    ; wieder an den Anfang gehen, wenn kein
                       Tastendruck erfolgte
- 18048 sta 1029     ; Bildschirmzeichen für <A> ausgeben
  18051 jsr 65490    ; CHROUT gibt ASCII-Zeichen aus

- 18054 bne 18043    ; zurück zum Anfang der Schleife
  ...               ; ...Abbruchbedingung ...
  ...
  18060 rts
```

Starten Sie nun das Programm, dann geht der Rechner erst einmal in Wartestellung, bis Sie eine Taste drücken.
Das entsprechende Zeichen erscheint dann auf dem Bildschirm, was noch von **GETIN** ausgeführt wird. Anschließend erscheint aber das Zeichen, das dem Bildschirmcode des (A)-Inhalts entpricht. Wenn genau das Zeichen ausgegeben werden soll, das der Taste entspricht, ist diese Art der Ausgabe nicht geeignet. Wir müssen dann z.B. **CHROUT** verwenden. Dann erfolgt der Ausdruck wie in BASIC mit CHR$(A).

Wenn Sie Schwierigkeiten haben, den ASCII-Code vom Bildschirm-Code zu unterscheiden, sollten Sie sich die entsprechenden Vergleichstabellen aus Ihrem Rechnerhandbuch bereitlegen und das eben besprochene Prográmmchen "60-getin" laufen lassen.
Wenn Sie eine Abbruchbedingung einbauen, kommen Sie sogar wieder aus der Schleife heraus:
Nehmen Sie z.B. die Taste "RETURN" als Abbruchtaste her, dann erzeugt das in (A) den Code 13. Vergleichen wir also mit diesem Wert und springen bei Gleichheit zum Befehl RTS.

7.2 Künstlicher Cursor mit GETIN und BSOUT

Wenn man die **GETIN**-Routine beispielweise in Textprogrammen verwenden will, sollte man auf jeden Fall wissen, an welcher Stelle man sich im Bildschirm gerade befindet.
Die einfachste Möglichkeit wäre es, das Cursor-AN/AUS-Flag zu sezten. Das hat aber den Nachteil, daß immer wieder einzelne Cur-

sorflecken beim Weiterschreiben stehenbleiben, was äußerst lästig ist.

Eine weitere Möglichkeit besteht darin, sich selbst einen Cursor zu basteln, der die momentane Position revers darstellt.
Dazu benötigen wir Zeile und Spalte, die ja stets in der Zeropage mitgeführt werden.

Zur Wiederholung:
Die Anfangsadresse der Zeile steht beim C64 in (209/210) bzw. beim 40/80XX in (196/197).
Die Spalte finden wir in (211) bzw. (198).

Ablauf:
01 Holen eines Zeichens in das (A)-Register
02 Die momentane Position auf jeden Fall nicht revers darstellen.
Dazu wird das Bit 7 des Codes mit AND #127 gelöscht:
Beispiel:

```
            <A> =  10110000   , also 176 (Code für rvs'0')
      AND #127     01111111
Ergebnis:   <A> =  00110000   , also  48 (Code für '0')
```

(AND erzeugt immer dann 1, wenn beide Bits 1 sind.)

03 Jetzt wird das geholte Zeichen ausgegeben. Damit erhöht sich auch die Cursorposition um 1.
04 Die nächste Position muß nun revers ausgegeben werden.
Das geschieht mit ORA #128, also durch Setzen von Bit 7.
Beispiel:

```
             <A> = 00110000   , also  48 (Code für '0')
       ORA #128    10000000
Ergebnis:    <A> = 10110000   , also 176 (Code für rvs'0')
```

(ORA erzeugt immer eine 1, außer wenn zwei Nullen miteinander verknüpft werden.)

ASSEMBLER-Beispiel "61-cursorget" (C64) :

```
01 18043 jsr 65508    ; GETIN holt ein Zeichen aus dem Tasta-
                        turpuffer nach (A)
   18046 beq 18043    ; Warten auf ein anderes Zeichen außer 0
   18048 tax          ; geholtes ASCII-Zeichen(!) nach (X) retten

02 18049 ldy 211      ; momentane Spalte holen
   18051 lda (209),y  ; Zeichen aus dem Bildschirm holen
   18053 and #127     ; in Normalcode umwandeln und gleich
```

```
   18055 sta (209),y ; wieder ausgeben

03 18057 txa         ; geholtes Zeichen wieder nach (A) holen
   18058 jsr 65490   ; und ausgeben mit CHROUT

04 18061 ldy 211     ; neue Spalte holen
   18063 lda (209),y ; neues Zeichen nochmal nach (A), diesmal
                       aber im Bildschirmcode
   18065 ora #128    ; und Zeichen in (A) mit Reversflag versehen
   18067 sta (209),y ; jetzt als Reverszeichen ausgeben

05 18069 bne 18043   ; unbedingter Sprung an den Schleifenanfang
```

Bitte beachten Sie:
Ein einfaches Erhöhen der Spaltenzahl z.B. mit INY wäre nicht sinnvoll, weil das eine Erhöhung bis 255 ermöglichen würde, während die normale Spaltenzahl nur 40 bzw. 80 beträgt.

Mit diesem Programm können Sie auf dem ganzen Bildschirm nach Herzenslust herummarschieren. Der künstliche Cursor zeigt Ihnen immer an, wo Sie sich gerade befinden. Allerdings blinkt er nicht wie gewohnt, sondern erzeugt ein stehendes Bild.

Falls Sie sich gewundert haben, warum wir im ASSEMBLER-Programm nicht gleich das geholte Zeichen mit STA(209),y ausgeben, sollten Sie sich noch einmal vor Augen führen, daß **GETIN** das geholte Zeichen auch gleich in den ASCII-Code umwandelt.
Dadurch können auch nicht druckbare Zeichen wie RETURN o.ä. verwendet werden.
Eine Rückumwandlung in den Bildschirmcode ist daher nicht angebracht.

Das obige Programm hat noch kein Zeichen als Abbruchkriterium definiert und läßt im übrigen alle Eingaben zu, also auch RUN/STOP.

Aufgabe:
Erweitern Sie das obige Programm, so daß sie mit der RETURN-Taste aussteigen können.

Aufgabe:
Verbessern Sie das Programm, so daß gleich nach dem Start ein Cursor erscheint und nicht erst nach Eingabe des ersten Tastendrucks.

7.3 Zahleneingabe (reell) mit GETIN und STRFAC

Um über die Tastatur Zahlen einzugeben und anschließend weiterverarbeiten zu können, übernehmen wir die Ziffern mit GETIN, legen sie zunächst als String ab und wandeln sie dann computergerecht in eine FLP-Zahl um.

Ablauf:

- Festlegen der Anfangsadresse des Strings, in den die Ziffern eingelesen werden sollen.
  Dazu verwenden wir zwei Zeropage-Adressen (LO/HI), um später die 'indirekt-indizierte Adressierung' anwenden zu können.
- Holen eines Zeichens mit **GETIN**.
  Anschließende Ausgabe mit **CHROUT**.
- Ablegen des Zeichens bis zum Endezeichen RETURN (Code 13).
- Code 13 durch Code 0 (Stringende) ersetzen.
- Umwandlung des Strings in eine FLP-Zahl.
- Weiterbehandlung z.B. Ausgabe, Abspeichern o.ä.

ASSEMBLER-Beispiel "62-getin-zahl" (C64):

```
- 18050 lda #0        ; LO der Stringanfangsadresse (256)
  18052 sta 60        ; in Zeropage
  18054 lda #1        ; dto. HI
  18056 sta 61        ; ablegen
        nop
- 18059 jsr 65508     ; GETIN holt ein Zeichen von Tastatur
  18062 beq 18059     ; warten bis Zeichen im Puffer
  18064 jsr 65490     ; Zeichen auf Schirm mit CHROUT
        nop
- 18068 ldy #0        ; Index für indirekte Adressierung
  18070 sta (60),y    ; Zeichen ablegen ab Adresse 256
  18072 inc 60        ; Adresse um eins erhöhen, also für die
                        nächste Ziffer vorbereiten
  18074 cmp #13       ; Ende der Eingabe erreicht?
  18076 bne 18059     ; nein ===> nächstes Zeichen holen

- 18078 dec 60        ; ja ---> Adresse wieder eins zurück
  18080 lda #0        ; und Byte 0
  18082 sta (60),y    ; statt Byte 13 ablegen (Endekriterium)
```

```
- 18084 lda 60        ; das ist gleichzeitig die Stringlänge, die
                        für STRFAC in (A) benötigt wird
  18086 ldx #0        ; LO des Stringanfangs nach (X)
  18088 stx 34        ; nach STRADR-LO
  18090 ldx #1        ; HI
  18092 stx 35        ; nach STRADR-HI
  18094 jsr 47029     ; STRFAC wandelt die Ziffernfolge ab
                        Adresse (256) in eine FLP-Zahl um
        nop
- 18098 jmp 43708     ; Weiterverarbeitung z.B. Ausdruck
```

Zur Wiederholung:
Der Befehl STA(60),y legt <A> in die Adresse ab, die sich aus der Summe <60/61> plus y errechnet. Also <60>+256·<61>+y.
In unserem Beispiel ist <60>=0 und <61>=1. Der Zähler y hat immer den Wert 0, so daß die erste Abspeicherung in 256 erfolgen muß.
Durch das Erhöhen von <60> mit INC 60 bereiten wir die nächste Abspeicherung vor.

Theoretisch sind jetzt 255 Ziffern möglich, was recht unsinnig ist und auch gefährlich. Denn ab (256) beginnt der Stackbereich, der zwar von oben nach unten volläuft, sich aber auf keinen Fall mit unserem String überschneiden darf.
Will man dem ganz aus dem Weg gehen, dann wählt man eben für diese Operationen einen anderen (ungefährdeten) Bereich im RAM aus. Wir haben das mit der indirekt-indizierten Adressierung schon ermöglicht.

Übrigens: Die Umwandlung erfolgt auch mit der Exponentialdarstellung (z.B. 0.5e-08) richtig.

Aufgabe:
Begrenzen Sie die Eingabe auf 10 Ziffern und operieren Sie in einem anderen Bereich.

Aufgabe:
Falls Sie sich bei der Eingabe der Ziffern vertippen sollten, haben Sie bis jetzt keine Korrekturmöglichkeit. Bauen Sie eine ein, indem Sie mit der Taste DEL nicht nur die Ziffer auf dem Bildschirm löschen, sondern auch das letzte Zeichen im String ungültig machen.

7.4 Eingabe mit BASIN

Die ROM-Routine **BASIN** ist eine recht universelle Eingabe-Routine, die wir weiter hinten immer wieder verwenden werden. Sie kann von allen Geräten Bytes aufnehmen und im (A)-Register bereitstellen.

Es lohnt sich daher, die Systematik dieses ROM-Programms näher zu untersuchen.

Wenn der Computer im sog. READY-Modus arbeitet, dann ist die Tastatur das Eingabe- und der Bildschirm das Ausgabegerät.

Ruft man nun **BASIN** auf, dann wird zunächst einmal nach dem aktiven Eingabegerät gefragt. Die Adresse **DEVIN** (153) bzw. (175) beim 80XX enthält den Wert Null, wenn die Eingabe über die Tastatur erfolgt.

Ist dies der Fall - und nur dann -, erfolgt ein Sprung zur Warteschleife, die auf einen Tastendruck lauert. Dort wird durch einen blinkenden Cursor, die aktuelle Bildschirmposition angezeigt und jede Tastatureingabe einschließlich aller Steuerzeichen auf dem Bildschirm ausgegeben.

Der Abschluß erfolgt erst dann, wenn die RETURN-Taste (Code 13) ausgelöst wurde. Jetzt wird die Bildschirmzeile, in welcher der Cursor gerade steht, auf Leerzeichen untersucht. Diese werden übersprungen und ein Bildschirmzeiger wird auf das erste Zeichen der aktuellen Zeile eingestellt. In den Akkumulator (A) kommt der Code 13. Ein Flag, belegt mit 13 zeigt an, daß die Eingabe beendet ist.

Ruft man nun wiederum **BASIN** auf, dann wird das erste Zeichen der Zeile eingelesen, in der man die RETURN-Taste gedrückt hatte. Weitere Aufrufe von **BASIN** holen Zeichen für Zeichen die Bytekette in das (A)-Register. Alle eingelesenen Zeichen werden dabei in den ASCII-Code umgewandelt, bevor sie im Akku abgelegt werden. Als letztes Zeichen wird wieder Code 13 übergeben. Daran läßt sich der Abschluß der Eingabe erkennen.

Man kann sich nun dieser Routine bedienen, wenn man über die Tastatur irgendwoher vom Bildschirm eine Zeile übernehmen will. Dabei wird komfortablerweise aus einem Bildschirmfenster immer

nur die Zeile bis zum rechten Fensterrand übernommen, wenn man sich mit dem Cursor vorher innerhalb des Fensters bewegt hat.

Die zu übernehmende Zeile braucht auch nicht vorher über die Tastatur geschrieben worden sein, sondern kann schon dort gestanden haben, bevor **BASIN** aufgerufen wurde. Erst nach dem Drücken der RETURN-Taste wird die momentane Cursorzeile festgehalten. Die übernommenen Zeichen können dann beliebig weiterverarbeitet werden.

Zweckmäßigerweise stellt man dafür einen Pufferplatz zur Verfügung, der nicht mehr als 80 Zeichen Umfang benötigt, weil eben nur zeilenweise eingelesen wird. Von diesem Puffer aus läßt sich nach dem Einlesen alles weitere unternehmen (z.B. Zuweisung auf eine Stringvariable, Umwandlung in eine Zahl oder andere Auswertungen).

Jetzt wird es Zeit für ein Beispiel, damit unsere Erklärungen auch nachprüfbar werden.

Ziel:
Eingabe eines Textes mit **BASIN**, Übernahme in den Bereich ab (10000) und Ausgabe der ersten 10 Zeichen dieses Strings.

Ablauf:
01 Zähler für die Ausgabe festlegen: (X)-Register
02 Aufruf von **BASIN**.
   Erst mit RETURN wird diese Routine abgebrochen. Der Bildschirmzeiger stellt sich auf das erste Element der Zeile, das kein Leerzeichen ist und holt es nach (A).
03 Zeichen aus (A) ab (10000) ablegen, wieder **BASIN** aufrufen, da jetzt das nächste Zeichen aus dem Bildschirm geholt wird, und mit erhöhtem Zähler bei 10000,x ablegen.
04 Solange Zeichen holen, bis der Code für RETURN auftritt, dann abbrechen.
05 10 Zeichen aus dem abgelegten String holen. Das entspricht der BASIC-Routine LEFT$(x$,10).

ASSEMBLER-Beispiel "63-basin" (C64):

```
01 11000 ldx #0       ; Zähler initialisieren
02 11002 jsr 65487    ; BASIN holt zunächst Zeichen aus der
                        Tastatur, nach RETURN aus dem Bilschirm
```

```
   11005 cmp #13      ; Code für RETURN ?
   11007 beq 11015    ; ja ===> zur Ausgabe
03 11009 sta 10000,x  ; nein ---> Zeichen ablegen
   11012 inx          ; Zähler erhöhen
04 11013 bne 11002    ; und unbedingt zu BASIN springen
05 11015 lda #16      ; Stringanfang LO
   11017 sta 34       ; nach STRADR-LO
   11019 lda #39      ; Stringanfang HI (von 10000)
   11021 sta 35       ; nach STRADR-HI
   11023 ldx #10      ; Länge nach (X)
   11025 jsr 43813    ; STROUT gibt linke Stringseite aus
   11028 rts
```

7.5 Eingabe einer Zeile mit INLINE

Eine ROM-Routine, die aus dem INPUT-Teil des BASIC-Systems entnommen ist, arbeitet sehr komfortabel, wenn es darum geht, ganze Zeilen über die Tastatur einzutippen, zu korrigieren und in eine Zeichenkette zu übertragen.
Es ist dies **INLINE**, das wiederum mehrfach das Unterprogramm **BASIN** (Eingabe BASIC-Text) aufruft.

Nochmals kurz zur Routine **BASIN:**
Ein Zeichen wird über die Tastatur nach (A) geholt und in den ASC-Code umgewandelt. Anschließend erfolgt die Ausgabe über den Bildschirm. Soweit besteht Identität mit **GETIN**, denn auch **BASIN** ist ein Teil von dieser Routine.
Allerdings werden solange Zeichen übertragen, bis die Taste RETURN gedrückt wird.

INLINE bedient sich durch ständiges Aufrufen von **BASIN** dieses Systems und wartet seinerseits ebenfalls auf die Taste RETURN. Wurde Sie gedrückt, so wird die Zeile, in der momentan der Cursor steht, vom Zeilenanfang bis zur letzten Spalte Zeichen für Zeichen aus dem Bildschirm-RAM gelesen und ab Adresse (512) ababgelegt.
Danach erfolgt der Abschluß der Eingabe durch Einsprung in eine weitere ROM-Routine (FININL), wo alle Flags, Zähler usw. wieder initialisiert werden. Unter anderem wird auch das Ende des Strings mit einer 0 (Byte 0!) markiert.

Gegenüber dem gewöhnlichen INPUT aus dem BASIC, haben wir hier noch den Vorteil, daß wirklich jedes Zeichen, das in der Zeile als gedrucktes Symbol darstellbar ist, auch in den String übernommen wird, was ja sonst nicht der Fall ist und mit einem 'extra ignored error' quittiert wird, wenn man z.B. ein Komma eingeben wollte.
Ein weiteres Plus dieser Routinenkombination ist, daß wir uns nicht um den Cursor kümmern müssen. Er ist hier mit eingebaut und zwar blinkend.

Die Anwendungsmöglichkeit der genannten Programmteile liegt auf der Hand: Immer wenn irgendwelche Texte über die Tastatur eingegeben werden sollen, bieten sich **INLINE, BASIN und GETIN** an.

Beispiel:
Eine beliebig beschriebene oder noch zu beschreibende Zeile soll per Cursorsteuerung mit RETURN übernommen werden.
Anschließend werden die einzelnen Zeichen des Strings ab (512) auf den Bildschirm gebracht.
Dabei läßt sich noch einmal der Unterschied zwischen dem ASC-Code und dem Bildschirmcode anschaulich vor Augen führen.

Ablauf:
01 Vorbelegen der Adresse (512) mit dem Byte 0, weil im Falle einer leeren Eingabe kein Endekriterium gefunden wird und ein 'string too long error' auftritt.
02 Holen der Zeichenkette bei gleichzeitigem Ausgeben der einzelnen Zeichen mit anschließendem Ablegen ab Adresse (512).
Das wird von der Routine INLINE alles auf einmal besorgt.
03 Ausgabe von "Schirm clear", wenn der Abschluß mit der Taste RETURN erfolgt ist.
04 Laden der abgelegten ASC-Codes und Ausgabe auf den Bildschirm mit STA XXXX.
05 Nochmaliges Laden und Ausgeben. Diesmal aber mit **CHROUT.**
06 Nochmaliges Ausgeben, diesmal aber mit der Routine **STR-0,** die alle Zeichen bis zum Erreichen eines 0-Bytes ausgibt.

Die Punkte 04 bis 06 sind lediglich zur Wiederholung und Übung eingebaut. Im "Ernstfall" wird hier die gewünschte Verarbeitungs-Routine stehen.

ASSEMBLER-Beispiel "64-inline-out" (C64):

```
01 18110 lda #0
   18112 sta 512     ; Endezeichen vorbelegen

02 18115 jsr 42336   ; INLINE holt eine ganze Zeile

03 18120 lda #147    ; Code für "Bildschirm clear"
   18120 jsr 65490   ; Schirm löschen
         nop
04 18124 ldx #0      ; Zähler initialisieren
   18126 lda 512,x   ; ein Zeichen des Strings holen
   18129 beq 18138   ; Endezeichen erreicht ===> weiter
   18131 sta 1024,x  ; sonst ab Bildschirmanfang (C64) ausgeben
   18134 inx         ; Zähler erhöhen
   18135 bne 18126   ; und zurück zum Schleifenanfang
         nop
05 18138 ldx #0      ; Zähler neu setzen
   18140 lda 512,x   ; ebenfalls ein Zeichen holen
   18143 beq 18151   ; falls Endezeichen in (A) ===> weiter
   18145 jsr 65490   ; sonst mit CHROUT ausgeben
   18148 inx         ; Zähler erhöhen
   18149 bne 18140   ; und weitermachen
         nop
06 18152 lda #13
   18154 jsr 65490   ; ein CR ausgeben (Code 13)

07 18157 lda #0      ; LO der Stringanfangs-Adresse
   18159 ldy #2      ; dto. HI (=512) nach (Y)
   18161 jsr 43806   ; Stringausgabe mit STR-O
   18164 clc         ; wenn eine neue Zeile geholt werden soll
   18165 bcc 18110
```

In einem Textprogramm wird man den String natürlich erst abspeichern, bevor man den nächsten holt.
Wie man ihn in eine BASIC-Variable übernimmt, behandeln wir demnächst (Kapitel 8).

7.6 Eingabe von Hexzahlen mit HEXINB und HEXINA (nur 40/80XX)

Aus dem Monitorteil der 80XX- und 40XX-Serien lassen sich zwei Routinen anzapfen, die Zahleneingaben im Hex-Format ermöglichen.

Dabei nimmt **HEXINB** genau zwei Zeichen, also ein Byte auf und überträgt es in das (A)-Register. Der Bereich ist dementsprechend beschränkt auf $00 bis $ff, das Dollarzeichen darf nicht mit eingegeben werden.

HEXINA übernimmt eine vierstellige Adresse in der gleichen Weise, legt aber das Ergebnis in (251/252) mit LO/HI ab.
Der Bereich entspricht dem Adressbereich von $0000 bis $ffff.

Dazu wieder ein kleines Anwendungsbeispiel:

Zunächst soll ein Byte in Hexform eingetippt und als Dezimalzahl ausgegeben werden. Anschließend geschieht das gleiche mit einer vierstelligen Adresse.

Ablauf und ASSEMBLER-Beispiel "65-hex-eingabe" (80XX):

```
01 jsr 55139      ; HEXINB wartet auf Eingabe einer zweistelli-
                    gen Hexzahl und abschließendem RETURN
02 tax            ; Byte als LOW nach (X)
   lda #0         ; 0 als HI nach (A), damit ist die Ausgabe einer
   jsr 53123      ; Integerzahl mit INTOUT vorbereitet

03 jsr 55124      ; HEXINA wartet auf Eingabe einer vierstelli-
                    gen Hexzahl mit abschließendem RETURN
04 ldx 251        ; LO aus (251) nach (X)
   lda 252        ; HI aus (252) nach (Y)
   jsr 53123      ; INTOUT druckt die Zahl dezimal
```

Eingabe-ROM-Routinen

Label	C64	40/80XX
GETIN	65508=$ffe4 KERNAL 1 Zeichen ---> (A)	65508=$ffe4
BASIN	65487=$ffcf KERNAL String ---> Schirm ---> einzeln nach (A)	65487=$ffcf
INLINE	42336=$a560 Zeile nach (512)... nimmt bis zu 80 Zeichen auf	46306=$b4e2
HEXINB	--- Byte in Hexeingabe ---> (A)	55139=$d767
HEXINA	--- Adresse(vierstellig) in Hexform ---> (251/252)	55124=$d754

8

Verwaltung der Variablen

8 Verwaltung der Variablen

In Maschinenprogrammen, die mit Variablen arbeiten, muß man entweder die Verwaltung darüber selbst kontrollieren - und das erfordert diszipliniertes Programmieren und Dokumentieren - oder aber man bedient sich der BASIC-Routinen und überläßt die Organisation dem Betriebssystem.
Insbesondere macht man sich dies bei der Verwendung von numerischen Variablen zunutze, wenn diese nach Abschluß eines Maschinenprogrammteils im BASIC-Programm weiterbehandelt werden sollen. Auch umgekehrt tritt der Fall häufig ein, daß nach einem BASIC-Vorspann, in dem etliche Variablen belegt werden, diese Werte in einem Maschinenteil verarbeitet werden sollen.

8.1 Überblick über die BASIC-Variablen

8.1.1 Lage der Variablen im RAM

Normalerweise werden die Variablen in der Reihenfolge, wie sie anfallen, anschließend an das BASIC-Programm angelegt, das ab 2048 (C64) bzw. 1024 (40/80XX) beginnt.
Ist kein BASIC-Teil vorhanden, so finden wir in den ersten drei Adressen (2048) bis (2050) jeweils eine 0. Die erste Variable wird also beim C64 ab (2051), beim 40/80XX ab (1027) aufgebaut.

In der Zeropage werden ständig Zeiger mitgeführt, die den Beginn und das Ende des Variablenbereichs (das ist auch gleichzeitig der Beginn der Felder) ausweisen.

Daraus folgt zunächst einmal, daß Maschinenprogramme in den oberen RAM-Bereich gelegt werden müssen, wenn die Variablenverwaltung von den BASIC-Routinen übernommen werden soll. Denn der Platzbedarf für die Variablen wächst von unten nach oben an. (Ausnahme bei den Strings, siehe unten!)

Beispiel:
Liegt ein Maschinenprogramm im RAM von (16000) bis (24000), so stehen für die Variablen die Adressen (2051) bis (15999) bzw. beim 40/80XX (1027) bis (15999) zur Verfügung.

Wer unbedingt sein Maschinenprogramm im unteren RAM-Bereich haben möchte, der muß die BASIC-Zeiger über seine letzte Maschinen-Adresse stellen.

8.1.2 Variablen-Arten

Im CBM-BASIC unterscheiden wir drei Variablenarten:
a) Realzahlvariablen (z.B. x1)
b) Integervariablen (z.B. x2%)
c) Stringvariablen (z.B. x3$)

Alle drei Arten können auch in Feldern verwendet werden. Doch diese Verwaltung wird kaum in Maschinenprogramme übernommen. Hier baut man besser eigene Strukturen auf. Beschränken wir uns daher zunächst auf einfache Variablen.

8.1.3 Struktur der Variablen

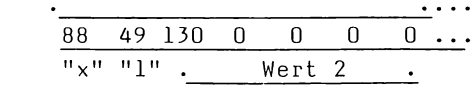
Alle Variablen belegen jeweils 7 Adressen im Speicher. Davon entfallen zwei auf den Variablennamen, die restlichen 5 Stellen werden unterschiedlich verwendet.

Reelle Zahlen:

Die ersten beiden Bytes enthalten den ASC-Code für den Variablennamen ohne jegliche Veränderung.
Werden nur einstellige Variablennamen verwendet, so wird das zweite für den Namen vorgesehene Byte mit 0 belegt.

Anschließend folgen die 5 Bytes für den codierten Wert der Zahl.

Beispiel:
Variablenname x1, Zahlenwert 2



Die reelle Zahl 2 wird durch die fünf Bytes 130/0/0/0/0 in Potenzschreibweise dargestellt, wobei das erste Byte den Exponenten durch entsprechendes Verschieben (Shiften) der nachfolgenden Man-

tisse angibt. Um recht/links unterscheiden zu können, geht man von 128 (=0 Shift) aus. Ist der Exponent nun größer als 128, dann erfolgt die Verlegung des Kommas nach rechts, ist er kleiner wird das Komma nach links versetzt.

In unserem Fall bedeutet 130, daß das Komma nach zwei Stellen erscheint. Im Dualzahlsystem sieht das so aus: 00000010,0000.... Das Bit 1, das hier plötzlich an der zweiten Stelle erscheint, wurde nirgendwo abgelegt, da es sich hier um eine positive Zahl handelt. Unser Computer "denkt" sich die Mantisse also zunächst so (er kann nur binär denken!): ,10000000... und verlegt dann das Komma um zwei Stellen nach rechts : 10,00000....

Die Mantisse besteht aus 32 (4 mal 8) Bits, wobei das höchste (ganz links) das Vorzeichen darstellt. Wenn es gesetzt ist, dann ist die Zahl negativ.

Die Zahl -2 sieht also dann so aus:

130 128 0 0 0 = -00000010,0000...

Beim sog. "Normalisieren" werden die Bits der Mantisse so lange nach links geshiftet, bis keine Vorkommanullen mehr auftreten.

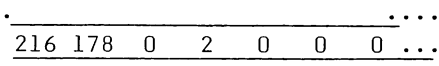
Das mag genügen. Man braucht sich beim Programmieren eigentlich nicht darum kümmern, wie die reellen Zahlen aufgebaut sind. Das kann unser Rechner besser und vor allem schneller.
Vielleicht eins noch: Die Nachkommastellen einer Dualzahl werden nach unten genau so in Zweierpotenzen weitergerechnet, allerdings mit negativem Exponenten.

$0{,}1111_2$ hat also den Wert 0+0,5+0,25+0,125+0,0675=0,9425

Integerzahlen:

In den ersten Bytes stehen wieder die Variablennamen. Damit man die Integerzahl von einer Realzahl unterscheiden kann, wird aber zu beiden ASC-Codes 128 addiert (Bit 7 gesetzt).
Anschließend folgt die Zahl in der Form HI/LO. Die restlichen drei Stellen werden immer mit 0 aufgefüllt.

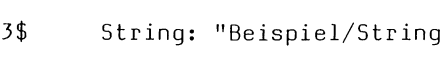
Beispiel:
Variablenname x2% Zahlenwert 2



Stringvariablen:

Bei den Strings geht es etwas komplizierter zu. In den ersten beiden Bytes steht ebenfalls der Variablenname, allerdings wird zur Unterscheidung nur im zweiten Namensbyte das Bit 7 gesetzt (ASC-Code + 128).
Anschließend folgt aber nicht etwa die Zeichenfolge des Strings, sondern nur eine Beschreibung, nämlich die Stringlänge (1 Byte) und die Anfangsadresse des Strings (2 Bytes LO/HI). Diese drei Bytes werden als String-Deskriptor bezeichnet.

Beispiel:
Variablenname x3$ String: "Beispiel/String"



Das bedeutet, daß der String 15 Zeichen lang ist und in unserem Fall ab Adresse (0+8·256)=(2048) mit dem Byte für "B" beginnt:

```
<0/8> = <2048> = 194  "B"
<1/8> = <2049> =  69  "e"
<2/8> = <2050> =  73  "i"
...     ...      ...
```

8.2 Einrichten einer Variablen

8.2.1 Festlegen des Bereichsanfangs

Wenn nicht mit einem kombinierten BASIC-Maschinenprogramm gearbeitet wird, empfiehlt es sich, den Variablenbereich zu Beginn eines reinen Maschinenprogramms selbst festzulegen.
Dazu setzen wir die Zeiger "Beginn der Variablen", "Beginn der indizierten Variablen", "Ende der Variablen" auf einen von uns gewählten Wert, der in einem RAM-Bereich liegt, der ansonsten von unserem Maschinenprogramm nicht berührt wird.

Beispiel:
Die Variablentabelle soll bei Page 10, also ab ($0a00)=(2560) beginnen.

ASSEMBLER-Beispiel "66-basic-zeiger" (C64 und 40/80XX):

.	C64		40/80XX	.
01	10000	lda #0		
	10002	ldy #10		
02	10004	sta 45	VARTAB-LO	sta 42
	10006	sty 46	VARTAB-HI	sta 43
03	10008	sta 47	ARYTAB-LO	sta 44
	10010	sty 48	ARYTAB-HI	sta 45
04	10012	sta 49	VAREND-LO	sta 46
	10014	sty 50	VAREND-HI	sta 47

Verwaltet man auch Stringvariablen, kann man das CBM-Konzept beibehalten, und die anfallenden Strings vom oberen RAM-Bereich herunterwandern lassen. Dazu muß aber das Maschinenprogramm abgesichert sein, was sich mit dem Zeiger für die RAM-Obergrenze erledigen läßt, nennen wir ihn **MAXMEM**. In erweiterten BASIC-Versionen existiert der Befehl HIMEM xxxx, mit dem man diese RAM-Obergrenze festlegen kann.

Beispiel (Fortsetzung):
Maschinenprogramm beginnt auf Page 40, also bei ($2800)=(10240).

.	C64		40/80XX	.
05	10016	lda #0		
	10018	ldy #40		
06	10020	sta 55	MAXMEM-LO	sta 52
	10022	sty 56	MAXMEM-HI	sta 53
		rts		

Werden nun Variablen eingerichtet, so beginnt die erste bei Adresse (2560) und alle weiteren können bis einschließlich (10239) aufgebaut werden. Eventuelle Strings "laufen" den Variablen ab (10239) nach unten entgegen. Bei Überschneidungen wird in BASIC ein OVERFLOW ERROR ausgegeben. In Maschinensprache hat man selbst dafür zu sorgen, daß die benötigten Bereiche sauber getrennt bleiben!

8.2.2 Suchen bzw. Einrichten einer Variablen mit PTRVAR

Die Routine **PTRVAR** durchsucht zunächst den festgelegten Variablenbereich. Dazu muß der Variablenname **VARNAM** in den Zeropageadressen (69/70) beim C64 bzw.(66/67) beim 40/80XX in der Reihenfolge LO=1.Variablenname, HI=2.Variablenname vorgegeben werden.

Ist die Variable bereits vorhanden, dann ist das Ergebnis ein Zeiger, der auf den Anfang der Variablen zeigt. Und zwar nicht auf den Variablennamen, sondern gleich dahinter auf den Inhalt. Bei Real- und Integerzahlen, wird also direkt auf den Anfang der Zahl selbst, bei Strings auf den Stringdeskriptor gezeigt. Beim C64 findet man diesen Variablen-Pointer **VARADR** in (71/72), beim 40/80XX in (68/69).

Wurde keine Variable gefunden, so wird hinter der letzten vorhandenen eine neue angelegt und mit Nullen aufgefüllt. Als Ergebnis erhält man wie oben den Zeiger hinter den Variablenkopf.

Außerdem ist in jedem Fall die Adresse der Variablen auch in den Registern (A/Y) mit LO/HI zu finden, so daß eine Weiterverarbeitung beschleunigt durchgeführt werden kann.

Beispiel:
Die bisher noch nicht vorhandene Variable x soll angelegt werden. Der Variablenbereich beginne bei (2580), also Page 10. Zur Überprüfung geben wir die gefundene Anfangsadresse der Variablen sowie den Variableninhalt auf dem Bildschirm aus.

Ablauf:
01 Setzen des Variablennamens **VARNAM**
02 Aufruf von **PTRVAR**
03 Retten des Variablenanfangs auf den Stack
04 Ausgabe der Variablenadresse **VARADR** als Integerzahl
05 Anfangsadresse der Variablen vom Stack holen und Ausgabe des Variableninhalts als reelle Zahl.

Anmerkung:
Der Inhalt von **VARADR** muß deswegen auf den Stack gerettet werden, weil durch die nachfolgende Ausgabe-Routine diese Zeropageadressen anderweitig verwendet werden. Da wir aber auch den Variableninhalt zur Kontrolle ausgeben wollen (Teil 05), legen wir den Inhalt von **VARADR** erst einmal auf den Stapel.

ASSEMBLER-Beispiel "67-vrb-zeiger":

```
01 lda #88          ; Code für "x"
   sta 69           ; Stelle für 1. Namenszeichen nach VARNAM1
   lda #0           ; Null, weil Variablenname nur einstellig ist
   sta 70           ; Stelle für 2. Namenszeichen nach VARNAM2
   Anmerkung: Nicht verwechseln mit der Variablen x0!

02 jsr 45287        ; PTRVAR sucht die Variable x, findet sie
                      nicht und legt sie neu an
03 lda 71           ; VARADR-LO
   pha              ; auf den Stack retten
   tax              ; und nach (X) übertragen (wegen INTOUT)
   lda 72           ; VARADR-HI
   pha              ; auf den Stack retten
04 jsr 48589        ; INTOUT druckt Anfangsadresse von x: 2582
05 pla              ; HI von VARADR holen
   tay              ; und nach (Y) übertragen
   pla              ; LO von VARADR nach (A) holen
   jsr 48034        ; MEMFAC holt Inhalt von x nach FAC
   jmp 43708        ; FLPOUT druckt x: 0
```

Anmerkung: Die Anfangsadresse 2582 wird nur dann ausgedruckt, wenn der Variablenanfang auf (2580) steht und x die erste verwendete Variable ist.

Im RAM-Bereich sieht das nun so aus:

```
    <2580> = 88  "x"
    <2581> =  0
    <2582> =  0
    <2583> =  0
    <2584> =  0
    <2585> =  0
    <2586> =  0
```

Die Variable "x" ist jetzt zwar angelegt, hat aber noch den Wert Null. Will man sie nun mit einer reellen Zahl belegen, so sind folgende Schritte notwendig:

```
06 Laden der gewünschten Zahl nach FAC1
07 <71> ---> <X>  LO der Anfangsadresse VARADR und
   <72> ---> <Y>  HI der Anfangsadresse übertragen nach (X/Y)
08 Aufruf der Routine FACMEM
```

Das Zurückholen aus dem Speicher geschieht entsprechend:

09 <71> ---> <A> LO der Anfangsadresse und
   <72> ---> <Y> HI der Anfangsadresse übertragen nach (A/Y)
10 Aufruf der Routine **MEMFAC**

Weitere Informationen zu diesem Thema finden Sie in Kapitel 10.

Label	C64	40/80XX
PTRVAR	45287=$b0e7	49543=$c187
	sucht Variable mit 1.Name/2.Name=<VARNAM>	
VARNAM1/2	(69/70)=($45/46)	(66/67)=($42/43)
	Variablennamel/2 für PTRVAR	
VARADR	(71/72)=($47/48)	(68/69)=($44/45)
	dort steht die Anfangsadresse LO/HI der Variablen nach Aufruf von PTRVAR	
TXTTAB	(43/44)=($2b/2c)	(40/41)=($28/29)
	Zeiger auf Beginn des BASIC-Textes	
VARTAB	(45/46)=($2d/2e)	(42/43)=($3a/3b)
	Zeiger auf Beginn der einfachen Variablen	
ARYTAB	(47/48)=$(2f/3a)	(44/45)=($3c/3d)
	Zeiger auf Beginn der indizierten Variablen (Felder=Arrays)	
VAREND	(49/50)=($3b/3c)	(46/47)=($3e/3f)
	Zeiger auf Ende der gesamten Variablentabelle	
MAXMEM	(55/56)=($37/38)	(52/53)=($34/35)
	Zeiger auf RAM-Obergrenze	

9

Bedienung der Peripherie

9 Bedienung von Peripherie

9.1 Datentransfer über den IEC- bzw. den seriellen Bus

Die Verbindung zwischen Computer und Peripherie wird bei den CBM-Geräten meist über den sog.IEEE-488- oder kurz den IEC-Bus hergestellt. Er mündet in eine 24-polige Steckverbindung aus, deren Leitungen zur Datenübertragung und -steuerung benötigt werden.

Der Datenbus ist bei den 40/80XX-Geräten 8 Bit breit, d.h. es können gleichzeitig 8 Bits übertragen werden. Damit kann also ein Byte in einem Takt übernommen (gelesen) oder übergeben (geschrieben) werden (parallele Schnittstelle).

Im Gegensatz dazu hat der C64 einen seriellen Bus, an dem gewöhnlich das Floppy angeschlossen ist. Die Daten werden hier wesentlich langsamer übertragen, weil hier Bit für Bit über die Leitung geschoben wird. Das übertragene Byte muß also zunächst zerlegt und im Empfänger wieder zusammengesetzt werden.
Wenn im folgenden Text von IEC-Bus gesprochen wird, so gilt sinngemäß beim C64 das gleiche für den seriellen Bus, weil beide analog arbeiten. Außerdem läßt sich der C64 auch auf den IEC-Bus umrüsten.

9.2 Umschaltungen des seriellen bzw. des IEC-Bus

Um die gewünschten Daten auch vom richtigen Gerät an die richtige Stelle zu übertragen, muß der Computer als "Controller" entprechende Schaltungen vornehmen.

- Dazu benötigt er in der Zeropage die Gerätenummer **GA** des angesprochenen Peripheriegeräts (GA heißt auch Primäradresse),

- eine Sekundäradresse **SA**, mit der ein Übertragungskanal freigesetzt wird

- und eine logische Adresse (auch File-Nummer oder Dateinummer) **LA**, mit deren Hilfe er Gerät und Kanal koordiniert.

Im einzelnen sind dies:

Label	C64		40/80XX
LA	184=$b8	logische Adresse	210=$d2
SA	185=$b9	Sekundäradresse	211=$d3
GA	186=$ba	Geräteadresse(Pr.)	212=$d4
GETSA	---		55599=$d92f

Wählt man die Sekundäradresse willkürlich, so besteht die Gefahr, daß der dadurch festgelegte Kanal nicht mehr frei ist. Will man dies vermeiden, so ruft man **GETSA** auf. Damit wird eine freie Sekundäradresse geholt und gleich nach **SA** gebracht. Das ist aber nur ab BASIC 4.0 möglich. Beim C64 besteht diese Variante leider nicht.

Zur Erinnerung:
Wenn keine Veränderungen vorgenommen wurden, gelten folgende **Gerätenummern** bei den CMB-Geräten:

0 = Tastatur
1 = Rekorder Nr.1
2 = RS232 über USER-Port bzw.40/80XX: Rekorder Nr.2
3 = Bildschirm
4 = Drucker
8 = Floppystation

Logische Adressen sind möglich von 1 bis 255, wobei ab 128 alle Ausgaben mit PRINT mit einem Linefeed (Zeilenvorschub) abgeschlossen werden.

Sekundäradressen wählen die Datenkanäle aus und können zunächst von 0 bis 31 gewählt werden. 0 und 1 sind aber z.B. beim Floppy reservierte Sekundäradressen für die Lade- und Speichervorgänge. Intern werden die Sekundäradressen für diverse Zwecke wieder verändert. Zum Beispiel wird aus der Sekundäradresse 0 beim Laden mit ORA #96 die Sekundäradresse 96 (siehe auch Beispiele unten).

Mit Hilfe der Sekundäradressen werden auch die einzelnen Betriebsarten der Peripheriegeräte - vorzugsweise der Drucker - angewählt. Das entsprechende Handbuch ist also hier zu Rate zu ziehen, damit es keine Fehlschaltungen gibt.

9.2.1 Datenübernahme mit TALK

Je nachdem, in welche Richtung der Datenfluß laufen soll, wird der Bus in einen besonderen Zustand versetzt.
Sollen Daten von außen - also von Peripheriegeräten - geholt werden, dann muß das entsprechende Gerät, ein Drucker oder das Floppy zum "Talker" (Sender) gemacht werden. Das geschieht mit der Routine **TALK**.

Nun kann über einen freien Kanal, der mit Hilfe der Sekundäradresse angewählt wird, die Übernahme der Daten erfolgen. Zu beachten ist, daß in der Zeropage eine Art Kontrollbyte mitgeführt wird, das eventuelle Fehler bei der Datenübertragung registriert. Es ist das **STATUS**-Byte oder kurz **STATUS**. Seine Adresse ist (144) beim C64 bzw. (150) beim 40/80XX.

Je nach Art des Fehlers wird ein Bit im Status gesetzt. Wichtigste Erkennung ist das Ende einer Datei. Hier wird Bit 5 gesetzt, **STATUS** hat also den Wert 64. Eine Übertragung kann nur erfolgen, wenn der **STATUS** Null ist.

Daraus ergibt sich für die Vorbereitung der Datenübernahme von einem Peripheriegerät folgende Sequenz:

ABLAUF für C64 (40/80XX in Klammern):

```
- lda #0                ; Null
  sta 144    (150)      ; für STATUS
- lda 186                ; Geräteadresse GA laden
  jsr 60681  (61650)    ; Routine TALK
- lda SA                 ; Sekundäradresse laden
  jsr 60871  (61763)    ; SASENT sendet die Sekundäradresse
```

Anschließend können über den nun freigelegten Kanal vorbereitete Daten übernommen werden (Beispiele siehe unten).
Nach Beendigung wird mit der Routine **UNTALK** der Bus wieder in den Wartezustand versetzt:

```
- jsr 60911  (61878)   ; UNTALK
```

9.2.2 Datenausgabe mit LISTEN

Analog zur eben besprochenen Datenübernahme ist die -ausgabe möglich:

ABLAUF für C64 (40/80XX in Klammern):

```
- lda #0                ; Null für
  sta 144     (150)     ; STATUS
- jsr 60684   (61653)   ; Routine LISTEN
- lda SA                ; Sekundäradresse laden
  jsr 60857   (61763)   ; und mit SASENL senden
..... Daten ausgeben ......
- jsr 60926   (61881)   ; Beenden mit UNLISN
```

Beim C64 ist zu beachten, daß es zum Senden der Sekundäradresse zwei verschiedene Einsprünge für **SASENX** gibt. Abhängig davon, ob die Routine **TALK** oder **LISTEN** vorausging, wird **SASENT** bzw. **SASENL** verwendet.

9.2.3 Beispiel: "68-druckaus" mit LISTEN, BSOUT, CLALL

Der Drucker kann als Empfänger (Listener) von Daten aktiviert werden, ohne daß ein direktes OPEN ausgegeben wird, wie es in BASIC z.B. mit OPEN 8,4,8 geschieht.
Allerdings wird neben der Gerätenummer auch eine logische Filenummer benötigt.

Die zum Drucker gesendeten Bytes laufen über den Bus zunächst in den Drucker-Puffer (falls vorhanden) und werden von dort aus ausgegeben.

Dabei ist zu beachten, daß einige Druckgeräte die Ausgabe auf Papier erst beginnen, wenn so viele Bytes empfangen wurden, bis eine Druckzeile voll wird, oder bis ein Steuerzeichen eintrifft wie z.B. Carriage Return (Code 13).

Sind alle Daten übertragen worden, wird der Drucker wieder deaktiviert mit **CLALL**, was auch **UNLISN** beinhaltet.

Im folgenden Beispiel bereiten wir den Drucker zur Ausgabe vor und geben Bytes aus, die wir ab Adresse (20050) bereitgestellt

haben. Als letztes Byte senden wir Code 13 für CR, damit die Ausgabe auf jeden Fall sofort erfolgt.
Die Byte-Leiste entspricht dem Text: "test-AUSDRUCK-123456"

Der Abschluß erfolgt mit **CLALL** (Erklärung siehe 9.2.9).

ASSEMBLER-Beispiel und Ablauf für "68-druckaus" (C64):

```
01 20000 lda #4       ; Gerätenummer für den Drucker z.B. 4
   20002 sta 154      ; in DEVOUT bereitstellen

02 20004 sta 186      ; Gerätenummer, auch 4 (=Drucker) nach GA

03 20006 lda #8       ; Sekundäradresse (hier 8)
   20008 sta 185      ; in SA bereitstellen
         nop
04 20011 lda #0       ; Null nach
   20013 sta 144      ; STATUS

05 20015 jsr 60684    ; LISTEN aktiviert das angesprochene Gerät
                        als Empfänger (hier den Drucker)

06 20018 lda 185      ; Sekundäradresse aus SA holen
   20020 jsr 60857    ; Sekundäradresse mit SASENL senden
   20023 nop
07 20024 ldx #0       ; Zähler für Ausgabe auf Null
   20026 lda 20050,x  ; Byte von vorbereiteter Leiste holen
   20029 jsr 65490    ; und mit BSOUT ausgeben
   20032 inx          ; Zähler erhöhen
   20033 cpx #21      ; letztes Zeichen erreicht?
   20035 bne 20026    ; nein ===> weiteres Byte holen und ausgeben

08 20037 jsr 65511    ; sonst Abschluß mit CLALL
   20040 rts          ; und Ende

Byteleiste ab (20050):
20050 B   84   69   83   84   45  193  213  211  196  210
20060 B  213  195  203   45   49   50   51   52   53   54
20070 B   13
```

Nach Aufruf mit SYS 20000 fängt der Drucker sofort an zu arbeiten und gibt anschließend ein CR aus.

Auf diese Art und Weise lassen sich natürlich auch die Steuercodes und alle Umschaltungen des Druckers bewerkstelligen. Ein Beispiel dazu finden Sie weiter hinten im dem Programm "printdirector", wo unter anderem auch Tabulatorsprünge programmiert werden.

Aufgabe:
Versuchen Sie, das Beispiel "druckerausgabe" so umzubauen, daß Sie Ihren Drucker als "Schreibmaschine" verwenden können. Jeder Tastendruck soll also sofort auf dem Drucker ausgegeben werden können.

Selbstverständlich ist das nicht sehr komfortabel, wenn man Texte schreibt, die man verbessern möchte. Aber für Mini-Schreibarbeiten ist das recht nützlich. Denken Sie nur daran: Wenn man ein Programmlisting ausgedruckt hat, möchte man gern mal etwas dazuschreiben ohne das Papier herauszunehmen oder das Progamm zu wechseln.

9.2.4 Anwendung: Modul "69-druckex"

Das folgende Modul entspricht in etwa der Lösung der eben gestellten Aufgabe. Aber es arbeitet noch etwas komfortabler: Nach dem Aufruf erhält man einen blinkenden Cursor wie im READY-Modus, mit dem man beliebig auf dem Bildschirm herumsausen kann. Auch alle anderen Cursorsteuerungen wie INST, DEL, CLR, HOME usw. sowie der Tabulator sind aktiv.

Nach dem Drücken der RETURN-Taste wird nun diejenige Zeile auf dem Drucker ausgegeben, in der der Cursor gerade steht. Dabei spielt es keine Rolle, ob der Text schon auf dem Bildschirm vorhanden war oder gerade erst geschrieben wurde.

Probieren Sie dieses Modul aus und Sie werden es ganz sicher in Ihr Textprogramm einbauen, denn es ist ideal zum Direktschreiben von der Computertastatur in den Drucker.

Dafür, daß der Drucker die richtige Schrift hat, müssen Sie selbst sorgen. Sie können diese Einstellungen vorher vornehmen oder aber das Modul "druckex" entsprechend erweitern.
(Siehe dazu auch die Beschreibung zum Modul "printdirector" im Abschnitt 9.10)
Haupt-ROM-Routine ist **BASIN**, die im Kapitel "Eingabe-Routi-

nen" erklärt wurde. Zur Ausgabe verwenden wir **LISTEN, BSOUT** und **UNLISN.**

Machen Sie sich bei dieser Gelegenheit noch einmal mit **BASIN** vertraut:

- **BASIN** holt so lange Zeichen von der Tastatur und gibt sie auf dem Schirm aus, bis RETURN auftritt.
- Anschließend wird das erste Zeichen der aktuellen Zeile in das (A)-Register geholt. Der Bildschirm ist nun das Eingabegerät.
- Neue Aufrufe von **BASIN** holen nun Zeichen für Zeichen aus dem Bildschirm-RAM.
- Nach dem letzten Zeichen steht 13 im (A)-Register.

Um einmal eine weitere übliche ASSEMBLER-Programmier-Art anzubieten, verwenden wir mal die reine LABEL-Schreibweise. Sie hat den Vorteil, daß man sehr schnell in ASSEMBLER programmieren kann. Man benötigt aber zum Eingeben auf diese Weise einen Editor, der die Labels aufnimmt und beim Assembliervorgang wieder in Zahlen und Adressen umsetzt.

ASSEMBLER-Programm "69-druckex" - Labelschreibweise:

MEMADR = Anfangsadresse eines freien RAM-Bereichs zum Zwischenspeichern der eingelesenen Zeile (max. Länge 80)
CR = 13 (RETURN)
DN = Gerätenummer für den Drucker
SEKADR = Sekundäradresse für Druckerbetrieb
ABBRUCH = beliebiger ASC-Code, mit dem das Programm beendet wird z.B. Klammeraffe

Alle anderen Labels sind besprochen und stehen in der Tabelle.

```
00 DRUCKER EIN   lda #4           ; evtl. Gerätenummer des Aus-
                 sta DEVOUT       ; gabegeräts bereitstellen
01 ANFANG        ldy #0           ; Zähler auf Null
02 EINGABE       jsr BASIN
                 cmp #ABBRUCH
                 beq 06 ENDE
                 cmp #CR
                 beq 03 EINGABENDE
                 sta MEMADR,y
                 iny
                 bne 02 EINGABE
03 EINGABENDE    sta MEMADR,y
```

```
                lda #DN
                sta LA
                sta GA
                lda SEKADR
                sta SA
                lda #0
                sta STATUS
                jsr LISTEN
                lda SA
                jsr SASENL
                ldx #0          ; Zähler auf Null
04 AUSGABE      lda MEMADR,x
                cmp #CR
                beq 05 ABSCHLUSS
                jsr OUTBUS (oder BSOUT)
                inx
                bne 04 AUSGABE
05 ABSCHLUSS    jsr UNLISN
                lda #CR
                jsr BSOUT       ; CR auf Bildschirm
                clc
                bcc 01 ANFANG   ; neue Zeile aufnehmen
06 ENDE         jsr UNLISN
                lda #3          ; evt. wieder Bildschirm als
                sta DEVOUT      ; Ausgabegerät bestimmen
                rts
```

Auch dieses Modul ist frei verschiebbar, kann also in jedem freien RAM-Bereich untergebracht werden. Es benötigt in der vorgestellten Form ca.80 Bytes Platz.

Bei der Ausgabe über den USER-Port kann es - je nach Programmierung des Druckers - notwendig werden, in Teil 01 zusätzlich das aktive Gerät in der Zeropage **DEVOUT** zu notieren:

```
                lda #4
                sta DEVOUT
```

Im letzten Teil 06 wird dann der Bildschirm als Gerät 3 wieder in Aktion gesetzt, bevor der Rücksprung erfolgt:

```
                lda #3
                sta DEVOUT
```

Schließt man die Übertragung mit **CLALL** (siehe unten) ab, dann wird automatisch der Bildschirm wieder als Ausgabegerät aktiviert. Die beiden gennannten Zeilen entfallen dann.

9.2.5 Vorbereitung von Datenübertragungen mit OPEN

Wenn Dateien angesprochen werden, dann erfolgt das von BASIC aus mit dem OPEN-Befehl. Die entsprechende ROM-Routine verlangt genauso folgende Parameter wie bisher beschrieben:

- logische Adresse in **LA**
- Geräteadresse in **GA**
- Sekundäradresse in **SA**

Zum Ansteuern einer bestimmten Datei (z.B. auf Diskette) werden jetzt noch benötigt:
- Anfangsadresse in **NAMADR** des Dateinamens mit LO/HI
- Länge des Dateinamens in **NAMLEN**

Sind die genannten Zeropageadressen belegt, dann kann die Routine **OPEN** aufgerufen werden.

9.2.6 Ausgabevorbereitung mit CHKOUT

Wir brauchen uns um die richtigen Schaltungen nicht zu kümmern, wenn wir die **OPEN**-Routine mit den eben genannten Parametern aufgerufen haben.

Soll eine Ausgabe auf die geöffnete Datei erfolgen, dann trifft die Routine **CHKOUT** die dazu notwendigen Vorbereitungen. Sie schaltet das Peripheriegerät auf "Empfang", indem sie das **STATUS**-Byte auf Null setzt, die Routine **LISTEN** aufruft und die richtige Sekundäradresse sendet.

Kurz: Das angesprochene Gerät wird als LISTENER aktiviert, so wie wir das im vorigen Abschnitt "zu Fuß" getan haben.
Wichtig dabei ist, daß **CHKOUT** nur dann wirksam werden kann, wenn die Datei geöffnet ist. Dann stehen nämlich in der sog. File-Tabelle die Parameter der angesprochenen Datei. Sie werden automatisch gefunden, wenn vor dem Aufruf von **CHKOUT** das X-Register mit der logischen Adresse derjenigen Datei belegt wird, die man ansprechen will.

Damit ergibt sich folgender Ablauf:

- Öffnen einer Datei auf das gewünschte Gerät mit **OPEN.**
- Übergeben der logischen Adresse LA ins (X)-Register, und Aufruf der Routine **CHKOUT.**
- entsprechende Ausgaben wie BASIC-PRINT, also mit **BSOUT** o.ä.
- Schließen der Datei mit **CLOSE.**

9.2.7 Eingabevorbereitung mit CHKIN

Entsprechend läßt sich der Bus auch auf 'Eingabe' schalten. Das bedeutet, daß alles was sonst über die Tastatur erwartet wird, nun vom IEC-Bus aufgenommen wird. Es ließe sich also z.B. eine weitere Tastatur oder ähnliches über den Bus bedienen.

Der Ablauf ist gleich wie oben. Lediglich die Routinen **CHKOUT** und **BSOUT** müssen durch **CHKIN** bzw. **BASIN** ersetzt werden.
Kurz: Das angesprochene Gerät wird als TALKER (Sender) aktiviert.

9.2.8 Standard-Ein/Ausgabe herstellen mit CLRCH

Um den Bus wieder in den Normalzustand zu versetzen, so daß die Tastatur (Gerät 0) das Eingabegerät und der Bildschirm (Gerät 3) das Ausgabegerät ist (wie z.B. im READY-Modus), springen wir die Routine **CLRCH** (Standardbus) an.
Eventuell noch geöffnete Dateien werden dabei allerdings nicht automatisch geschlossen. Sie können aber geöffnet bleiben. Bei mehreren Hin- und Herschaltungen des Busses muß man dabei selbst den Überblick behalten.
Zweckmäßigerweise öffnet man eine Datei zu Beginn eines Programms und schließt sie erst vor dem Rücksprung in das aufrufende Programm, so daß bei der Unterprogrammtechnik keine 'file not/open'-Fehler auftreten können.

9.2.9 Dateien schließen mit CLALL

Auch bei der Umschaltung mit CLRCH (clear chanels) bleiben die Dateien offen. Um sämtliche noch offene Files auf einmal ordnungsgemäß abzuschließen, wird die Routine **CLALL** aufgerufen. Wir haben sie vorhin schon ein paarmal verwendet.

9.2.10 Schließen einer Datei mit CLOSEA und CLOSEL

Entsprechend dem BASIC-CLOSE werden nicht mehr benötigte Dateien abgeschlossen, wenn man das (A)-Register mit der logischen Adresse belegt und anschließend die ROM-Routine **CLOSE** aufruft. Verwaltet man nur eine einzige Datei und hat man die Zeropage-Adresse **LA** (siehe oben) nicht verändert, dann kann man sich das Laden ersparen und springt einen Befehl weiter hinten im ROM ein bei **CLOSEL**. Beim C64 ist dies allerdings nicht möglich.

Die CLOSE-Befehle bewirken auf jeden Fall, daß die zugeordneten Fileparameter in der Filetabelle gelöscht werden.
Daraus folgt, daß nach dem Schließen einer Datei die Vorbereitungsroutinen **CHKIN** oder **CHKOUT** nicht mehr richtig arbeiten können. Sie werden erst wieder nach dem erneuten Öffnen der gewünschten Dateien wirksam.

9.3 Vereinfachungen zur Dateibehandlung

Wenn man in einem Programm mehrere Dateien offen hält, um sie im Wechsel zu bearbeiten, kann man bei den CBM-Rechnern bis maximal 3 Files (8OXX:1O) geöffnet lassen.

Jedesmal, wenn ein Gerät angesprochen wird, müssen dafür Geräte- und Sekundäradresse sowie die logische Adresse gesetzt werden. Nach dem OPEN-Befehl führt der Rechner aber selbständig Tabellen für jede noch offene Datei und kann selbständig Gerätenummer und Sekundäradresse bereitstellen, die er anhand der logischen Adresse findet.

Dazu stellt man die logische Adresse LA im (A)-Register bereit und ruft zunächst die Routine **SUFTAB** (suche Filedaten in der Tabelle) auf.
Anschließend wird durch **SETTAB** (setze Tabellendaten) das Bereitstellen von logischer Adresse (LA), Geräteadresse (GA) und Sekundäradresse (SA) in die dafür vorgesehenen Zeropage-Adressen besorgt.
Nach wie vor muß aber noch das über die geöffnete Datei angesprochene Gerät als "Talker" oder "Listener" aktiviert werden.

Das eben besprochene Verfahren ist dem BASIC-System entnommen.

Erinnern Sie sich: Um z.B. den Drucker anzusprechen, befehlen Sie OPEN 2,4,8. Später schreiben Sie nur noch z.B. PRINT #2,..., geben also nur noch die logische Adresse (File-Nummer) an. Der Rechner findet nun das richtige Gerät (Nummer 4) und den richtigen Befehlskanal (über Nummer 8).
Und genau diese Such- und Setzroutinen **SUFTAB** und **SETTAB** verwenden wir für unsere ASSEMBLER-Programme auch.
Entsprechendes gilt natürlich auch für die Benützung der Routinen **CLOSEX.**

Hier zunächst die wichtigsten ROM-Routinen und Adressen:

Label	C64	40/80XX
NAMLEN	183=$b7	209=$d1
NAMADR	187/188=$bb	218/219=$da/db
STATUS	144=$90	150=$96
OPEN	62282=$f34a	62819=$f563
SUFTAB	62223=$f30f	61245=$f2c1
SETTAB	62239=$f31f	62157=$f2cd
CLOSEA	62097=$f291	62178=$f2e2
CLOSEL	---	62176=$f2e0

Gemeinsame KERNAL-Routinen:

CHKOUT	65481=$ffc9
CHKIN	65478=$ffc6
CLRCH	65484=$ffcc
CLALL	65511=$ffe7

9.4 Behandlung von Dateien

Verarbeitet werden zwei Arten von Dateien: die sog. sequentiellen Dateien, wo die Bytes in laufender Folge geschrieben oder gelesen werden und die sog. relativen Dateien, die einen wahlfreien Zugriff auf eine beliebige Stelle innerhalb des Files ermöglichen. Programme fallen unter die Rubrik sequentiell. (Siehe auch 9.5.2)

Außerhalb dieser von BASIC vorgegebenen Dateien lassen sich in Maschinensprache beliebige andere Strukturen erstellen, was sich aber für den Hausgebrauch selten lohnt.

Zur Vereinfachung der Programmierarbeiten empfiehlt es sich, das File-Konzept des Betriebssystems beizubehalten. Damit läuft die Dateibehandlung entsprechend der BASIC-Logik ab:

- Öffnen der Datei
  mit logischer Adresse **LA**, Geräteadresse **GA**, Sekundäradresse **SA** und dem Dateinamen **FILNAM.**
- Datentransfer:
  Eingabe(Lesen) oder Ausgabe(Schreiben)
- Schließen der Datei

In den folgenden Abschnitten schauen wir uns an Hand eines Beispiels diese drei Punkte genauer an und erklären dabei den Einsatz der ROM-Routinen.

9.5 Arbeiten mit SEQ-Dateien

9.5.1 Öffnen einer SEQ-Datei

Um eine Datei ansprechbar zu machen, müssen folgende Werte übergeben werden:

- Logische Adresse **LA** (frei zwischen 1 und 255)
- Sekundäradresse **SA** (von 2 bis 31)
  Nummer 15 stellt den sog. Befehls-(oder Kommando-)kanal dar, über den die Befehle an das Disk-Operating-System gesandt werden.
  0 und 1 dagegen sind reserviert für Laden und Speichern.
- Gerätenummer **GA**
- Dateiname bzw. seine Länge **NAMLEN** und Anfangsadresse **NAMADR**

Wählt man die Sekundäradresse willkürlich, so besteht die Gefahr, daß der dadurch festgelegte Kanal nicht mehr frei ist. Will man dies vermeiden, so ruft man **GETSA** auf. Damit wird eine freie Sekundäradresse geholt und gleich nach **SA** gebracht (nicht C64).

Beim Dateinamen gilt aber zu beachten, daß er nicht nur aus dem

eigentlichen Namen besteht, sondern immer einen Zusatz enthalten muß, der auf die Art des Zugriffs hinweist. Das ist deshalb erforderlich, weil das DOS (Disk-Operating-System) sonst nicht weiß, wie die Datei zu behandeln ist.

Bei den SEQ-Dateien sind folgende Angaben möglich:

"w" (Code 87) für erstmaliges Schreiben in die Datei (Anlegen)
"a" (Code 65) für Weiterschreiben (wie BASIC APPEND)
"r" (Code 82) für Lesen (read)

Zu beachten ist, daß eine bereits vorhandene Datei nicht noch einmal mit "w" anzusprechen ist, wenn sie beschrieben werden soll, sondern immer mit "a".

Der Name muß also folgende Form haben (Beispiel für SEQ-Datei mit der Bezeichnung "test"):

"0:test,w" für Anlegen der Datei und erstmaliges Schreiben
"0:test,a" für fortlaufendes Weiterschreiben, wobei die neuen Daten an die bestehenden angehängt werden.
"0:test,r" für Lesen aus der Datei, wobei immer von Beginn der Datei an gelesen wird.

Auf diese Weise wird allerdings immer das Laufwerk 0 angesprochen. Will man auf Laufwerk 1 zugreifen, dann muß vor den Namen eine 1 und ein Doppelpunkt gesetzt werden. Laufwerk 0 läßt sich ebenfalls auf diese Weise kennzeichnen.

Beispiel: "1:test,r" bedeutet Laufwerk 1, Datei "test" zum Lesen

Nachdem nun der Dateiname noch weitere Angaben enthält, die der Floppy mitgeteilt werden müssen, spricht man nun von einem Befehlsstring. Kürzen wir ihn ab mit **BF1**.

Dieser Befehlsstring wird in BASIC erst einmal zusammengebaut aus den Parametern der BASIC-Zeile und danach ab (850) abgelegt. Wir können ihn aber an jede beliebige RAM-Stelle schreiben und haben ihn somit jederzeit griffbereit. Er wird dann auch nicht überschrieben, wenn z.B. eine andere Datei angesprochen wird.

Beispiel: Befehlsstring anlegen

Wir bereiten einen Befehlsstring ab (20580=$5084) vor mit der Form: "0:test,w"

```
- 20580   48 "0"
  20581   58 ":"
- 20582   84 "t"
  20583   69 "e"
  20584   83 "s"
  20585   84 "t"
- 20586   44 ","
  20587   87 "w"
```

Damit hat er die Anfangsadresse NAMADR=(100/80) und die Länge NAMLEN=8

ASSEMBLER-Beispiel "70-seq-open": Öffnen der Datei "test"
(Adressen für C64)

```
- 20500 lda #5      ; logische Adresse LA
  20502 sta 184     ; nach LA
- 20504 lda #6      ; Sekundäradresse 6 (Beispiel!)
  20506 sta 185     ; nach SA
- 20508 lda #8      ; Geräteadresse 8 (Floppy)
  20510 sta 186     ; nach GA
- 20512 lda #100    ; LO von Befehlsstring-Anfang
  20514 sta 187     ; nach NAMADR-LO
  20516 lda #80     ; HI von Befehlsstring-Anfang
  20518 sta 188     ; nach NAMADR-HI
- 20520 lda #8      ; Länge des Befehlsstrings
  20522 sta 183     ; nach NAMLEN
- 20524 jsr 62282   ; OPEN öffnet die Datei
  20527 rts
```

9.5.2 Beispiel: Schreiben mit CHKOUT und BSOUT

Ol Um in die Datei zu schreiben, müssen die entsprechenden Kanäle "freigelegt" werden. Das besorgt die ROM-Routine **CHKOUT.** Sie bereitet den IEC-Bus als Listener vor, sorgt dafür, daß das richtige Gerät und die richtige Datei angesprochen werden und die Daten den richtigen Weg nehmen.

Voraussetzung dafür ist, daß die gewünschte Datei (noch) offen ist.
Die ROM-Routine **CHKOUT** benötigt im (X)-Register die logische Adresse (Filenummer). Mit deren Hilfe kann sie aus einer in der Zeropage geführten Tabelle die richtigen Zuordnungen treffen.

02 Die Routine **BSOUT**, die wir schon als **CHROUT** kennengelernt haben, ist vielseitig einsetzbar. In unserem Fall gibt sie das im (A)-Register befindliche Zeichen über den aktiven Kanal, also über den Bus an das Floppy aus.

03 Als Abschluß müssen die Kanäle wieder in den Standard-Zustand versetzt werden, was die Routine **CLRCH** besorgt. Damit ist wieder die Tastatur als Talker und der Bildschirm als Listener aktiv.

ASSEMBLER-Beispiel: "7l-seq-write" (C64 und 8OXX):

```
01 20530 ldx #5       ; Filenummer (zu obigem Beispiel)
   20532 jsr 6548l    ; CHKOUT Ausgabe-Vorbereitung
02 20535 ldy #4       ; Zähler (Beispiel) für 4 Datenbytes
   20537 lda 2058l,y  ; Laden eines Zeichens
   20540 jsr 65490    ; BSOUT gibt Zeichen auf BUS
   20543 dey          ; weiterzählen
   20544 bne 20537    ; Ende erreicht? nein ---> weitere Ausgaben
03 20546 jsr 65484    ; ja ---> CLRCH Standard-Ein-Ausgabe
   20579 rts          ; wieder herstellen
```

In unserem Beispiel geben wir das Wort "test", das wir vorher abgelegt haben, als Datenkette aus.

Voraussetzung für das Funktionieren dieses Teils ist natürlich, daß der Öffnungsteil aufgerufen wurde.

9.5.3 Beispiel: SEQ-Lesen mit CHKIN und BASIN

Um aus der Datei "test" zu lesen, muß sie - wie oben erwähnt - mit dem Befehlsstring "l:test,r" geöffnet worden sein.

01 Die Routine **CHKIN** bereitet entsprechend die Eingabe vor. Die Voraussetzungen sind die gleichen wie bei **CHKOUT**.

02 **BASIN** holt über den aktiven Kanal ein Zeichen in das (A)-Register.

03 Als Abschluß erfolgt wieder der Aufruf von **CLRCH.**

ASSEMBLER-Beispiel "72-seq-read" (C64 und 8OXX):

```
01 20552 ldx #5       ; Filenummer nach (X)
   20554 jsr 65478    ; CHKIN bereitet Eingabe vor
02 20557 ldy #3       ; Zähler auf 3 (Beispiel für 3 Daten)
   20559 jsr 65487    ; BASIN holt Datenbyte nach (A)
   20562 sta 899,y    ; Byte ablegen
   20565 dey          ; weiterzählen
   20566 bne 20559    ; Ende erreicht? nein ===> weiter einlesen
03 20568 jsr 65484    ; ja ---> CLRCH Standardzustand
04 20571 lda #132     ; LO von (900)
   20573 ldy #3       ; HI von (900)
   20575 jsr 43806    ; STR-O gibt den String aus: "est"
  (20575 jsr 47901)
   20578 rts
```

Wir haben die Daten in unserem Beispiel gleich weiterverarbeitet und zur Kontrolle ausgedruckt. Das Nullbyte als Abschluß des Strings ist zunächst vorhanden, wenn ab (900) vorher keine anderen Operationen stattgefunden haben.

9.4.4 Beispiel: Schließen der Datei mit CLOSEA oder CLOSEL

Wird die Datei nicht mehr gebraucht, muß sie geschlossen werden. Dazu benötigen wir wieder die logische Filenummer, diesmal aber im (A)-Register. Anschließend erfolgt der Aufruf von **CLOSE.**

ASSEMBLER-Beispiel: "73-close file" (C64 und 8OXX):

```
- 20590 lda #5       ; Filenummer von "test"
  20592 jsr 62097    ; CLOSEA
 (20592 jsr 62178)
  20595 rts
```

oder - wenn das aktuelle File aus **LA** geschlossen wird:

```
- 20590 jmp 62176    ; CLOSEL  (nicht für C64 geeignet)
```

9.5.5 Verknüpfen der SEQ-Routinen

Wenn Sie alle vier vorangegangenen ASSEMBLER-Programme eingegeben haben, können wir zum Ausprobieren schreiten.
Zunächst erstellen wir eine kleine Sprungleiste zum Neu-Anlegen der SEQ-Datei "test". Erinnern Sie sich: Das "w" muß hinter den Dateinamen gesetzt werden?

ASSEMBLER-Beispiele "74-seq-rout" (allgemein):

```
01 20600 lda #87      ; Code für "w"
   20602 sta 20587    ; in den Befehlsstring einbauen
   20605 jsr 20500    ; Unterprogramm "65-seq-open"
   20608 jsr 20530    ; Unterprogramm "66-seq-write"
   20611 jmp 20590    ; Unterprogramm "67-close file"
```

Rufen Sie SYS 20600 auf. Sie erkennen am Inhaltsverzeichnis, daß die sequentielle Datei "test" angelegt worden ist.

Zum Weiterschreiben gehen wir ähnlich vor, allerdings ist das Zeichen "a" im Befehlsstring erforderlich:

```
02 20615 lda #65      ; Code für "a"
   20517 sta 20587    ; einbauen
   20520 jmp 20605    ; und zum Schreibprogramm springen
```

Nach dem Aufruf wird noch einmal das Wort "test" in die Datei "test" geschrieben, so daß jetzt enthalten ist: "tsettset", weil wir beide Male mit dem letzten Buchstaben zu schreiben angefangen haben.
Wiederholte Aufrufe (SYS 20615 o.ä.) schreiben immer wieder diese vier Zeichen an die bereits bestehenden.

Jetzt kommt die Probe auf das Exempel: Wenn alles glatt gegangen ist, können wir nun auch aus der Datei wieder alles oder einen Teil herauslesen, wie es uns beliebt.

```
03 20625 lda #82      ; Code für "r"
   20627 sta 20587    ; in den Befehlsstring einbauen
   20630 jsr 20500    ; Unterprogramm "file open"
   20633 jsr 20552    ; Unterprogramm "seq-read"
   20636 jmp 20590    ; Unterprogramm "close file"
```

Nach dem Aufruf erscheinen die Buchstaben "est", wenn "seq-read" unverändert übernommen wurde.
Probieren Sie nun mal die Programme durch mit verschiedenen Namen und Zeichenlängen. Verwenden Sie auch mal Trennzeichen beim Schreiben und beobachten Sie die Ausgabe!

Wenn Sie wollen, können Sie auch Programme auf diese Weise abspeichern. Allerdings sind solche Programmdateien mit "seq" gekennzeichnet und können auch nicht mit LOAD oder DLOAD geladen werden, weil der Zeiger am Anfang der Datei fehlt. (Siehe dazu Kapitel 11).

Man kann allerdings solche maskierten Programme mit Hilfe der Lese-Routine an eine beliebige Stelle laden und dann als Programm aufrufen. Solche "üblen" Tricks sind oft Ansatzpunkte zum Schutz gegen allzu rasche Einsicht in ein Programm.

Versuchen Sie also auch mal das Speichern und Laden auf diese Weise!

9.6 Arbeiten mit REL-Dateien - Schreiben/Lesen mit BSOUT/BASIN

Den Vorteil, daß man auf jeden Record einer Datei - und hier wieder auf jede Position - zugreifen kann, können wir auch in Maschinensprache wahrnehmen.

9.6.1 REL-Dateien auf dem C64

Da die meisten C64-Besitzer noch nie mit REL-Dateien gearbeitet haben, weil das BASIC 2.0 dies nicht so ohne weiteres vorsieht, hierzu die wichtigsten Erläuterungen:

REL-Dateien lassen sich mit Karteikästen vergleichen. Jede Karte (Record) ist nummeriert (Recordnummer) und hat Platz für eine bestimmte Menge an Einträgen (Recordlänge = Anzahl Bytes/Record). In jedem Karteikasten (REL-Datei) befinden sich nur Karten von gleicher Größe (Recordlänge). Dabei ist es unerheblich, ob und wieviele Einträge auf den Karten vorgenommen wurden. Der Inhalt jeder Karte (Record) kann beliebig oft beschrieben, gelesen oder

verändert werden. Auch neue Karten können aufgenommen werden (max. 65535 Records).

Der Vorteil dieser REL-Dateien ist, daß wir unserem "Sekretär" Aufträge erteilen können wie:

Lies vor, was in Karte (Record) 10, ab Buchstabe (Byte) 20 steht!
oder
Ändere das 23.Zeichen von Record 12: statt "w" schreibe "r"! usw.

Wenn wir wollen, können wir mit Hilfe von Trennzeichen (z.B. Komma, Strichpunkt, CR, Doppelpunkt usw.) jede Zeichenfolge in einzelne Wörter zerlegen oder das Ende eines Records mit einem Zeichen (z.B. 255) markieren und damit die Records strukturieren.

In SEQ-Dateien ist letzteres zwar auch möglich, jedoch besteht die ganze Datei praktisch nur aus einem einzigen Satz - allerdings von beliebiger Länge. Beim Lesen muß man immer wieder ganz von vorn anfangen, beim Schreiben kann man lediglich hinten etwas dranhängen (Appendix=Anhang). Fehler lassen sich nicht ohne weiteres ausbessern. Man kann höchstens die gesamte Datei noch einmal abgeändert anlegen.

Damit erkennen wir den Vorteil der REL-Dateien sofort: Es ist der Zugriff nach freier Wahl, sowohl beim Schreiben als auch beim Lesen. Der Nachteil: Es wird mehr Platz auf der Diskette beansprucht.
Die Verwendung von REL-Dateien liegt auf der Hand: Immer dann, wenn Daten häufig geändert oder ergänzt werden müssen und die einzelnen Datensätze durchnummeriert sein sollen.

Für REL-Dateien gilt:
- Jeder Record kann max. 254 Zeichen fassen.
- Es sind max. 65535 Records möglich.

Das BASIC des C64 kennt zwar keine Befehle zur REL-Datei-Verwaltung. Jedoch kann das am meisten benützte Floppy 1541 durchaus REL-Dateien verarbeiten.
Mit ein paar Vorbereitungen sollte es uns also gelingen, die notwendigen Befehle zu erarbeiten und richtig anzuwenden.

Im Gegensatz zu seinen größeren Brüdern gibt es beim C64 ein paar Punkte mehr zu beachten:

- Der C64 kann jeweils nur eine REL-Datei geöffnet halten. Bevor mit einer zweiten gearbeitet wird, muß die erste ordnungsgemäß abgeschlossen worden sein.
  Das heißt, daß wir die logische Adresse nicht variabel halten müssen. Wir legen sie für unsere Arbeit hiermit willkürlich auf Nr.5 fest.

- Die Befehlsstrings 1 und 2 haben das gleiche Format wie in 9.5.1 beschrieben. Sie sollten aber beim C64 über zwei OPENs ausgegeben werden.

  Der 2. Befehlsstring muß aber unbedingt über den Befehlskanal Nr.15 (Sekundäradresse #111) gesendet werden.
  Als logische Adresse verwenden wir in unserem Beispiel Nr.2.

- Beide Eröffnungen (mit LA=5 und LA=2) sollten erst beim Abschluß aller Schreib/Lese-Vorgänge geschlossen werden. Wird der Befehlskanal zu früh geschlossen, kann es Floppy-Probleme geben.

- Zur Ein- bzw. Ausgabe der Bytes verwenden wir die Routinen CHKIN/BASIN bzw. CHKOUT/BSOUT. Ein Zergliedern wie wir es bei dem REL-Beispiel für den 80XX getan haben, um möglichst viel Zeit einzusparen (STATUS,LISTEN,SASEND,OUTBUS...) ist beim C64 mit etwas mehr Aufwand verbunden und spart daher so gut wie keine Zeit mehr ein.

- Als Abschluß der Ein/Ausgaben verwenden wir CLRCH.

- Während das Einrichten einer REL-Datei mit BASIC 4.0 und den großen Floppys automatisch mit dem Beschreiben des ersten Records problemlos abläuft, benötigt der C64 dazu eine Extrawurst.

  Die einzelnen Datensätze, eben die sog. Records mit ihrer festen Länge, müssen auf der Diskette erst einmal angelegt werden - wie das Einlegen leerer Karten in einen Sortierkasten. Das geschieht dadurch, daß man in den voraussichtlich letzten Record das Byte 255 schreibt. Das DOS übernimmt als braver Sekretär den Rest und baut die gesamte Datei auf.

  Man sollte daher allerdings im voraus wissen, wieviele Records die Datei einmal fassen soll. Ein paar Reserve-Records schaden beim Einrichten nicht, benötigen aber Platz auf der Diskette,

auch wenn sie noch nicht beschrieben wurden.

Nehmen wir an, wir wollen 100 Sätze zu je 40 Zeichen verwalten. Dann schreiben wir in den Record #100 eine 255 in das 1.Byte. Der Platzbedarf beträgt dann 100 mal 40 = 4000 Bytes. Das macht 16 Blöcke plus 1 Verwaltungsblock. Auf dem Inhaltsverzeichnis der Diskette wird diese REL-Datei demnach mit 17 Blöcken Platzbedarf ausgewiesen, egal ob bereits darauf geschrieben wurde oder nicht.

Braucht man später mehr Platz, kann man die REL-Datei auch über den letzten Satz hinaus weiterbeschreiben. Das kostet aber immer ziemlich viel Zeit. Deshalb lohnt sich das Einrichten mit einer genügend hohen Anzahl von Records.

- Der Abschluß wird immer mit je einem CLOSE auf die logischen Adressen der Datei (Nr.5) und des Befehlskanals (Nr.2) vorgenommen.

Weil die Verwaltung von REL-Dateien von wirklich praktischer Bedeutung ist, beschreiben wir das entsprechende ASSEMBLER-Programm für den C64 hier sehr ausführlich.

Insgesamt gliedert sich ein Programm für REL-Dateien in folgende Teile:

01 Ausgeben des Befehlsstrings Nr.1 über die logische Adresse der REL-Datei und einer beliebigen freien Sekundäradresse.
In unserem Beispiel: LA=5, SA=5

02 Ausgeben des Befehlsstrings Nr. 2 über eine andere logische Adresse und die Sekundäradresse 15 für den Befehlskanal.
In unserem Beispiel: LA=2, SA=15

03 Schreiben in die REL-Datei entsprechend dem 2. Befehlsstring, der u.a. Recordnummer und Byteposition enthält.
z.B. BF2 = 80 101 20 0 12, also wird Record #20, Byte #12 als Anfangsposition festgesetzt, wobei die Datei mit der Sekundäradresse #101 (96+5) angesprochen wird.
(80 ist der Code für "p", der das Positionieren einleitet.)

04 Lesen aus der REL-Datei entsprechend dem 2. Befehlsstring analog zum Schreiben 03.

Die eingelesenen Bytes werden zur weiteren Verwendung erst einmal in einen freien Bereich abgelegt.

05 Schreibroutine zum Anlegen einer neuen REL-Datei.
Sie setzt in den angegebenen letzten Record das Byte 255 auf Platz 1 und löst damit das Einrichten der gesamten Datei aus.

z.B. BF1="0:reldatxx,l,40" - Laufwerk, Name, Länge
BF2= 80 101 100 0 1 - "p", SA, Record #100, Byte #1

06 Abschließen der Dateien durch Ausgeben von CLOSE auf die beiden logischen Adressen.
in unserem Beispiel: CLOSE#5, CLOSE#2

Als Beispiel für das Funktionieren unseres Programms fügen wir noch an:

07 Stringausgabe für die eingelesenen Bytes bis zum Auftreten eines Bytes mit Inhalt Null.

08 Routine zum Übertragen der beiden Befehlsstrings 1 und 2 an die im Programm verwendete Stelle: (900) bis (926)

09 Eine Byte-Leiste, die die beiden Befehlsstrings für unser Beispiel enthält und zwar zunächst in der Form, wie wir sie zum Neueinrichten einer REL-Datei brauchen (siehe Punkt 05).

Alle Programmteile 01 bis 09 benötigen einschließlich etlicher Orientierungs-NOPs nicht einmal 250 Bytes und passen daher auf einen einzigen Diskettenblock.

Anmerkung:
Wir haben im folgenden Beispiel die Länge des Befehlsstrings variabel gehalten, weil der Dateiname ja zwischen 1 und 16 Zeichen lang sein kann. Die jeweils gültige Länge setzen wir in die Zeropageadresse 158 und rufen sie von dort aus ab.

Wer sich diesen Luxus schenken will, kann auch mit der vorgegebenen Länge von 8 Zeichen für den Dateinamen arbeiten. Auch damit lassen sich mehr REL-Dateien angeben, als man jemals in seinem Leben auf Diskette bringen wird.

Übrigens: Beim Ausprobieren des REL-Programms blinkt die Floppy wie bei einem Fehler, wenn Sie eine REL-Datei neu anlegen. Lassen

Sie sich dadurch nicht verwirren, der nächste Befehl wird einwandfrei angenommen.

ASSEMBLER-Beispiel "75-reldateien" (C64):

01 Befehlsstring 1 mit OPEN ausgeben - REL-Datei öffnen

```
30000 nop
30001 lda #5       ; Sekundäradresse 5
30003 sta 185      ; nach SA
30005 nop          ; GETSA ist beim C64 nicht möglich
30006 lda #8       ; Geräteadresse 8 (Floppy)
30008 sta 186      ; nach GA
30010 lda #5       ; logische Adresse
30012 sta 184      ; nach LA
30014 ldx 158      ; vorbereitete Länge des 1.Befehlsstrings holen
30016 stx 183      ; nach NAMLEN
30018 lda #132     ; Adresse LO des 1.BF
30020 sta 187      ; nach NAMADR-LO
30022 lda #3       ; Adresse HI des 1.BF
30024 sta 188      ; nach NAMADR-HI
30026 jmp 62282    ; OPEN gibt 1.Befehlsstring aus
      nop
```

02 Befehlsstring 2 mit OPEN ausgeben - Recordzeiger setzen

```
30030 lda #5
30032 sta 183      ; Länge des 2.Befehlsstrings nach NAMLEN
30034 lda #2
30036 sta 184      ; logische Adresse #2
30038 lda #15      ; Sekundäradresse 15 für Kommandokanal
30040 sta 185      ; nach SA
30042 lda #154     ; Anfang LO von BF2
30044 sta 187      ; nach NAMADR-LO
30046 lda #3       ; Anfang HI von BF2
30048 sta 188      ; nach NAMADR-HI
30050 jmp 62282    ; Ausgabe von BF2 als Befehl mit OPEN
   .. nop ..
```

03 Daten in Record schreiben - Beispiel

```
30055 ldx #5       ; logische Adresse der REL-Datei nach (X)
30057 jsr 65481    ; und damit CHKOUT aufrufen
30060 ldy #0       ; Zähler auf 0 für Byteausgabe
```

```
30062 lda 19900,y ; ein Byte aus Bereich ab (19900) holen (Bsp.)
30065 beq 30075   ; beim Auftreffen auf Endezeichen 0 ===> Sprung
30067 jsr 65490   ; BSOUT gibt Zeichen auf vorbereiteten
                    Kanal aus
30070 iny         ; weiterzählen
30071 bne 30062   ; und weiterausgeben
30073 beq 30080   ; falls Zähler 0 ===> keine Ausgabe mehr
30075 lda #255    ; als letztes Zeichen Byte 255
30077 jsr 65490   ; mit BSOUT ausgeben
30080 jsr 65484   ; und Abschluß mit CLRCH
   .. nop ..
```

Anmerkung: Die Ausgabe eines Null-Bytes als Abschluß wird nicht akzeptiert. Deswegen haben wir als Endekriterium 255 in die REL-Datei bzw. den Record gesetzt.

04 Lesen aus einem Record - Beispiel

```
30085 ldx #5      ; mit logischer Adresse
30087 jsr 65478   ; und CHKIN Eingabe vorbereiten
30090 ldy #0      ; Zähler
30092 jsr 65487   ; ein Byte nach (A) mit BASIN holen
30095 sta 10000,y ; und vorläufig ablegen
30098 bmi 30103   ; wenn Code größer 128, also spätestens beim
                    Auftreten von 255 ===> aufhören
30100 iny         ; sonst weiterzählen
30101 bne 30092   ; und einlesen
30103 lda #0      ; Byte Null
30105 sta 10000,y ; als Endekriterium des eingelesenen Strings
30108 jmp 65484   ; und Abschluß mit CLRCH
```

Anmerkung: Das Nullbyte setzen wir als Endezeichen, damit die später folgende Stringausgabe mit **STR-O** erfolgen kann. Die Begrenzung der Codenummern auf 127 ist hier willkürlich gewählt und kann selbstverständlich abgeändert werden.

05 Schreiben in den letzten Record zum Neueinrichten

```
30130 ldx #5      ; logische Adresse
30132 jsr 65481   ; CHKOUT
30135 lda #255    ; Byte 255 nach (A)
30137 jsr 65490   ; und mit BSOUT ausgeben
30140 jmp 65484   ; CLRCH
```

06 Schließen der Dateien

```
30115 lda #5      ; logische Adresse der REL-Datei
30117 jsr 62097   ; CLOSEA
30120 lda #2      ; logische Adresse der "Kommandodatei"
30122 jmp 62097   ; CLOSEA
```

Anmerkung: Die "Kommandodatei" ist nur zum Zwecke der Ausgabe des Befehlsstrings 2 eröffnet worden.

07 Stringausgabe

```
30150 lda #16     ; LO von Stringanfang 10000
30152 ldy #39     ; HI von Stringanfang 10000
30154 jmp 43806   ; String ausgeben mit STR-O
```

Anmerkung: Die Stringausgaben kann man auch verwenden, um REL-Dateien zu beschreiben, also bei Teil 03.

08 Befehlsstringübertragung

Beispiele für die Befehlsstrings stehen ab (30220) bis (30246) und werden nach (900) bis (926) übertragen.

```
30207 ldx #26     ; Zähleranfang
30209 lda 30220,x ; Zeichen aus Byte-Leiste holen
30212 sta 900,x   ; und in Kassettenpuffer übertragen
30215 dex         ; weiterzählen
30216 bpl 30209   ; bis Zähler 0: übertragen
30218 rts         ; Rücksprung
```

09 Byte-Leiste dazu:

```
30220   B    48   58   82   69   76   68   65   84   88   88
30230   B    44   76   44   40    0    0    0    0    0    0
30240   B     0    0   80  101  100    0    1
```

In diesem Fall ist die Byte-Leiste so hergerichtet, daß damit die REL-Datei "reldatxx" mit der Recordlänge 40 mit einem Gesamtumfang von 100 Records neu angelegt werden kann.

Stellen wir uns nun mit Hilfe von Sprungleisten aus diesen 9 Programmteilen die drei REL-Verwaltungsprogramme zusammen:

R1 Einrichten der Datei "reldatxx" mit "76-rel einricht":

Voraussetzung: Länge des 1.Befehlsstrings: z.B. <158>=14

```
30160 jsr 30207  ; Unterprogramm 08 "Befehlsstringübertragung"
30163 jsr 30000  ; U 01 "BF1 ausgeben"
30166 jsr 30030  ; U 02 "BF2 ausgeben"
30169 jsr 30130  ; U 05 "letzten Record schreiben"
30172 jmp 30115  ; U 06 "close"
```

R2 Schreiben in die Datei "reldatxx" mit "77-rel write":
Voraussetzung: beide Befehlsstrings im Kassettenpuffer
Länge 14 in (158)
Vorbereitung: Befehlsstring 2 nach Bedarf abändern

```
30175 jsr 30000   ; U 01
30178 jsr 30030   ; U 02
30181 jsr 30055   ; U 03 "Record beschreiben"
30184 jmp 30115   ; U 06
```

R3 Lesen aus der Datei "reldatxx" mit "78-rel read":
Voraussetzung: wie R2
Vorbereitung: wie R2

```
30190 jsr 30000   ; U 01
30193 jsr 30030   ; U 02
30196 jsr 30085   ; U 04 "Record lesen"
30199 jsr 30115   ; U 06 "close"
30202 jmp 30150   ; U 07 "Stringausgabe"
```

Anmerkung: Die beiden Unterprogramme 01 und 02 können auch zu einem einzigen zusammengefaßt werden, dazu ist der erste JMP-Befehl durch einen JSR-Befehl zu ersetzen.

Wie man sich solcher Programme auch von BASIC aus bedienen kann, wird in den letzten Kapiteln an Hand von Anwendungen erklärt.

Probieren Sie aber zunächst einmal an den eben vorgestellten Beispielen Ihre eigenen Versionen, Verbesserungen und Erweiterungen aus.

9.6.2 REL-Dateien auf 4Oer- und 8Oer-Geräten

Die "großen" CBM-Systeme verlangen dabei folgendes Vorgehen, wie es auch bei der BASIC-Sequenz "DOPEN .... DCLOSE" sichtbar wird: (Für den C64 gelten etwas andere Spielregeln. Siehe dazu den vorhergehenden Abschnitt 9.6.1!)

Ablauf "rel-dateien":

01 Initialisierungen, Sekundäradresse bereitstellen

02 Gerätenummer GA, Filenummer (logische Adresse) LA, Adresse und Länge des Dateinamens NAMADR und NAMLEN bereitstellen und Ausgabe der Filedaten über den IEC-Bus an das Floppy.
Dieser Befehlsstring muß dabei folgende Daten enthalten:
- Laufwerknummer, Doppelpunkt
- Dateiname ohne Anführungszeichen
- Komma, Art der Datei (hier ein "l" für Recordlänge)
- Komma, Byte für Recordlänge (1 bis 255)

Mit diesen Angaben wird die Datei eröffnet oder angelegt. Die Initialisierung (Teil 01) setzt die gewünschten Werte.

03 Position des RECORD-Zeigers über den IEC-Bus an das Floppy senden. Diese Bytefolge enthält immer 5 Zeichen:
- Code für "p", also 80 (positionieren)
- Sekundäradresse, die zur geöffneten Datei gehört
- RECORD-Nummer LO
- RECORD-Nummer HI
- Byte-Nummer im Record (1 bis 255)

Diese Ausgabe erfolgt über den Befehlskanal mit der Sekundäradresse #111 (aus #15 ora #96).

04 Daten schreiben oder lesen über die Sekundäradresse der geöffneten Datei (Übergabe an den IEC-Bus).
Diesmal verwenden wir dazu die Routinen **OUTBUS bzw.INBUS**, die nicht über den Sprungverteiler laufen und damit etwas weniger Zeit benötigen.
Auch die Ausgabevorbereitung erledigen wir diesmal selbst und erkennen dabei gleichzeitig, welche Arbeit uns die etwas langsamere Routine **CHKOUT** abnimmt:
STATUS auf Null, **LISTEN** ausgeben, Sekundäradresse senden.

05 Abschluß durch Deaktivieren des IEC-Bus und schließen der Datei.

Beispiel: Schreiben in die REL-Datei "reldat01"

Wir nehmen folgende Werte an:
- Recordlänge l=40
- Schreiben in Satz 1, ab Byte 20
- Logische Adresse der Datei: 1
- Laufwerk 1
- Weiterhin nehmen wir an, daß die Bytefolge aus Teil 02 (Laufwerk, Name usw,) im 2.Bandpuffer ab (900=$0384) bereits eingerichtet ist (1. Befehlsstring **BF1**).
- Auch die Bytefolge zur Positionierung des Recordzeigers (siehe Ablauf 03) hat ihren festen Platz ab (922=$039a). (2.Befehlsstring **BF2** mit 5 Zeichen)

Die Inhalte der beiden Befehlsstrings können sie rasch über BASIC oder Maschine verändern:

- Laufwerknummern Drive 0: <900>=48, Drive 1: <900>=49
- Dateiname: ab (902) bis max.(917)
  anschließend die Folge: 44,76,44,Recordlängenbyte
- Eingabe der RECORD-Länge (905+Länge des Namens)
- Eingabe der RECORD-Nummer <924/925>=LO/HI
- Bytenummer im RECORD (926)

Für das folgende ASSEMBLER-Beispiel gilt:
- Die Länge des 1.Befehlsstrings muß nach (180) gebracht werden!
- Die logische Adresse muß in (182) stehen!

In BASIC würden unsere Befehle so aussehen:
```
dopen#1,"reldat01",140,d1:record#1,1,20:print#1,"abc...":dclose#1
```

Der erste Befehlsstring (entspricht dopen#1,"reldat01",140,d1):

```
ASC-Zeichen:       1  :  r  e  l  d  a  t  0  1  ,  1  ,  (
ASC-Code:         49 58 82 69 76 68 65 84 48 49 44 76 44 40
Lage: ab      (900)          .....                    (913)
Länge: hier 14, max. 22
```

Der zweite Befehlsstring (entspricht record#1,1,20):

```
ASC-Zeichen:       p  " (nd)(nd)(nd)     nd=nicht druckbar
ASC-Code:         80 98   1   0  20
Lage:            (922) bis (926)
Länge: immer 5
```

Die Sekundäradresse 98 ergibt sich aus #2 ora #96.

Anmerkung: Die beiden Befehlsstrings müssen nicht im Bandpuffer liegen. Sie können zum Beispiel auch direkt vor oder nach dem Hauptprogramm liegen, was zwar eine bessere Kompaktheit bietet, aber unser Modulkonzept stört (beim Verschieben in einen anderen Bereich).
Im folgenden Beispiel schreiben wir in die REL-Datei Daten, die wir zuvor ab Adresse (19900) bereitgestellt haben. Es sind alphanumerische Zeichen, die uns später das Testen unserer Programmteile erlauben, weil sie geordnet sind:

"0123456789:abcdefghijklmnopqrstuvwz"

Dieser String erscheint im Teil 01 als Byte-Leiste.

ASSEMBLER-Programm "79-record/w 80"(nur 80XX/40XX):

```
01 19900 B  48   49   50   51   52   53   54   55   56   57
   19910 B  58   65   66   67   68   69   70   71   72   73
   19920 B  74   75   76   77   78   79   80   81   82   83
   19930 B  84   85   86   87   88   89   90    0    0    0

   19992 lda #14     ; Länge von BF1
   19994 sta 180     ; nach (180) - für unser Beispiel
   19996 lda #1      ; logische Adresse für unser Beispiel
   19998 sta 182     ; nach (182) - zur Wiederverwendung
   20000 jsr 55599   ; GETSA holt freie Sekundäradresse

02 20003 lda #8      ; Floppy-Nummer
   20005 sta 212     ; nach Geräteadresse GA
   20007 lda 182     ; vorbelegte logische Adresse LA holen
   20009 sta 210     ; nach LA bringen
   20011 ldx 180     ; gegebene Länge des Befehlsstrings holen
   20013 stx 209     ; und nach NAMLEN
   20015 lda #132    ; Befehlsstring1-Anfangsadresse LO
   20017 sta 218     ; nach NAMADR-LO
   20019 lda #03     ; Befehlsstring1-Anfangsadresse HI
   20021 sta 219     ; nach NAMADR-HI
   20023 jsr 62819   ; OPEN
   20026 nop
03 20027 ldx #5      ; feste Länge von Befehlsstring2
   20029 stx 209     ; nach NAMLEN
   20031 lda #154    ; Befehlsstring2-Anfangsadresse LO
```

```
   20033 sta 218      ; nach NAMADR-LO
   20035 lda #03      ; Befehlsstring2-Anfangsadresse HI
   20037 sta 219      ; nach NAMADR-HI
   20039 lda 210      ; logische Adresse LA holen
   20041 jsr 62145    ; SUFTAB sucht Filedaten in Tabelle
   20044 jsr 62157    ; SETTAB setzt die Tabellendaten
   20047 lda 211      ; holt Sekundäradresse für File
   20049 sta 923      ; in Befehlsstring2 einbauen
   20052 lda #111     ; Sekundäradresse 111 (Kommandokanal)
   20054 jsr 55963    ; Befehlsstring2 mit BFOUT ausgeben

04 20057 lda #0       ; Status
   20059 sta 150      ; zurücksetzen
   20061 jsr 61653    ; LISTEN
   20064 lda 923      ; Sekundäradresse des Files holen
   20067 jsr 61763    ; mit SASEND ausgeben
   20070 ldx #20      ; Zähler für Byte-Ausgabe
   20072 lda 19899,x  ; Byte von Daten-Leiste laden (Beispiel !)
   20075 jsr 61854    ; und mit OUTBUS ausgeben
   20078 dex          ; Zähler erniedrigen
   20079 bne          ; Zähler ungleich 0 ---> weiterausgeben

05 20081 jsr 61881    ; UNLISN
   20084 lda 210      ; Filenummer LA
   20086 jsr 62176    ; CLOSE schließt die Datei
   20089 rts          ; Rücksprung zur aufrufenden Routine
```

Noch ein paar Hinweise:
Bevor dieses Modul ab (20000) aufgerufen wird, muß die Länge des ersten Befehlsstrings in (180) und die logische Adresse (File-Nummer) in (182) vorliegen.
Wollen Sie zum Ausprobieren unseres Beispiel die vorgegebenen Werte (NAMLEN=14 und LA=5) beibehalten, dann rufen Sie SYS 19992 auf.

Denken Sie auch daran, daß die beiden Befehlsstrings ab (900) bzw. ab (922) zuvor eingerichtet sein müssen? Von BASIC aus kann man das mit einer DATA-Routine o.ä. bewerkstelligen. Wenn sich der Dateiname nicht mehr ändert, installieren Sie ihn fest ab Adresse (902).

In (20070) bis (20079) werden hier im Beispiel 20 Zeichen ausgegeben, die ab (19900) stehen. Dieser Teil ist den jeweiligen Erfordernissen anzupassen.

Wenn Sie von BASIC aus das dopen#la,"name",d(lw),l(rl) durchführen, springen Sie in (20027) ein. Führen Sie auch den Befehl record#(la),rn,rb vorher aus, springen Sie bei (20057) ein. Das Schließen der Datei wird anschließend immer besorgt.

Schauen Sie sich dazu auch im vorigen Abschnitt das Beispiel für den C64 an. Es bietet weitere Varianten.

Falls Sie die Datei erst bei Bedarf schließen wollen, bauen Sie ab (20084) die Abfrage eines CLOSE-Flags ein und überspringen den letzten Teil, wenn es nicht gesetzt ist.

Mit dem folgenden BASIC-Programm können Sie die Funktionsfähigkeit des Moduls "record/write" testen.

Von Zeile 100 bis 950 findet das Initialisieren statt.

Rufen Sie dann das Modul mit SYS 20000 auf. Es erzeugt die RELdatei "reldat01" und schreibt den Record Nummer 1 ab Byte 20.

Mit RUN 1000 können Sie den Inhalt wieder lesen.

BASIC-Testprogramm "reltest 80":

```
100 rem beispiel zu modul record/w 80
200 rem befehlsstring 1 nach (900)...
300 for n=0to13:read bf:poke900+n,bf:next n
400 rem filenummer und namlen poken
500 poke 182,1: rem file
600 poke 180,14 : rem länge des befehlsstrings 1
700 rem befehlsstring 2 nach (922)...
800 for n=0 to 4:read bf: poke922+n,bf: next n
900 data 49,58,82,69,76,68,65,84,48,49,44,76,44,40
910 data 80,98,1,0,20
950 end
1000 rem test
1100 dopen#1,"reldat01",d1:record#1,1,20:input#1,x$:printx$
1200 dclose#1: end
```

Lesen aus einer REL-Datei "80-record/r80":
Um das Programm zum Lesen umzufunktionieren, müssen Sie nur den Teil 04 ändern:

```
04 ...
+  20061 jsr 61650    ; TALK
   20064 lda 923      ; Sekundäradresse holen
   20067 jsr 61763    ; und mit SASEND ausgeben
   20070 ldx #0       ; Zähler 0 setzen
   20072 jsr 61888    ; Byte vom IEC-Bus holen mit INBUS
   20075 sta 24000,x  ; in freiem Bereich ablegen (Beispiel !)
   20078 inx          ; Zähler erhöhen
   20079 cpx #20      ; letztes Byte eingelesen? (Beispiel !)
   20081 bne 20072    ; nein ---> weiterlesen

05 20083 jsr 61878    ; UNTALK
  ...   usw.
```

Die restliche Verwaltung erfolgt wie oben beschrieben.

Aufgaben:
Probieren Sie diese Programmteile mit anderen Dateinamen und anderen Recordzeigern aus.

Zerlegen Sie das ASSEMBLER-Beispiel in die Teile:
Öffnen, Zeiger setzen, Schreiben, Lesen, Schließen
und erstellen Sie JSR-Leisten, mit denen Sie nach Belieben eine REL-Datei bearbeiten können. (Siehe dazu auch Abschnitt 9.6.1!)

Adressen und ROM-Routinen zur Dateiverwaltung

Label	C64	40/80XX
NAMLEN	183=$b7	209=$d1
NAMADR	187/188=$bb	218/219=$da/db
LA	184=$b8	210=$d2
GA	186=$ba	212=$d4
SA	185=$b9	211=$d3
BF1	900=$0384 Anfangsadresse des 1.Befehlsstrings	
BF2	922=$039a dto. des 2.Befehlsstrings (Beispiel!)	
GETSA	-----	55599=$d92f
OPEN	62282=$f34a	62819=$f563
SASENL	60857=$edb9	61763=$f143
SASENT	60871=$edc7	61763=$f143

SUFTAB	62223=$f30f	61245=$f2c1
SETTAB	62239=$f31f	62157=$f2cd
BFOUT	--- (mit OPEN)	55963=$da9b
LISTEN	60684=$ed0c	61653=$f0d5
OUTBUS	60893=$eddd	61854=$f19e
UNLISN	60926=$edfe	61881=$f1b9
TALK	60681=$ed09	61650=$f0d2
INBUS	60947=$ee13	61888=$f1c0
UNTALK	60911=$edef	61878=$f1b6

CHKOUT	65481=$ffc9
BSOUT(CHROUT)	65490=$ffd2
CHKIN	65478=$ffc6
BASIN(CHRIN)	65487=$ffcf
CLRCH	65484=$ffcc

CLOSEA	62097=$f291	62178=$f2e2
CLOSEL	----	62176=$f2e0

Für den C64 gilt zu beachten, daß es zwei verschiedene Routinen zum Ausgeben der Sekundäradresse gibt: die erste (LI) wird nach einem LISTEN gesendet, die zweite nach einem TALK (TA).

9.7 Laden eines Programmes mit LOAD oder LOADXX

Ohne das Öffnen der Datei mit **OPEN** lassen sich sowohl Programme wie auch Dateien laden, wenn man **LOAD** oder **LOADXX** verwendet. Dabei spielt es keine Rolle, ob es sich um BASIC- oder Maschinenprogramme handelt. (**LOAD** ist nicht für den C64 geeignet.) Nehmen wir an, auf Diskette steht die Datei "bastest", die mit "prg" gekennzeichnet ist. Dann läßt sich dieses Programm wie folgt laden:

Beispiel:
Öffnen von "bastest" wie in 9.4.1 beschrieben bis **OPEN.** Nun werden **STATUS** und das sog. Load/Verify-Flag **LVFLAG** auf Null gesetzt.

Wird **LVFLAG** mit 1 belegt, erfolgt nur das "Verifying", also die Überprüfung, ob das abgespeicherte Programm mit dem geladenen übereinstimmt.

Ablauf:

```
01....06   wie 9.4.1 (Adressen für 40/80XX)
07 lda #0        ; Null nach
   sta 150       ; STATUS
   sta 157       ; LVFLAG auf "laden" stellen
08 jsr 62472     ; LOAD holt das Programm "bastest" in den
                   Arbeitsspeicher
```

Der Programmanfangs- und der Programmendezeiger werden dabei auf den mit dem Programm abgespeicherten Anfang gesetzt. Bei den 80XX-Geräten werden also (40/41) und (42/43) neu eingestellt.

Ist dies unerwünscht, weil man z.B. aus einem BASIC-Programm heraus ein Maschinenmodul nachladen möchte und anschließend mit der nächsten BASIC-Zeile weiterfahren will, so springt man die Routine **LOADXX** an, die sämtliche Zeiger unverändert läßt.

Das folgende Beispiel ist sowohl für C64 als auch für 80XX-Geräte geeignet (40/80XX-Adressen in Klammern):

```
01...07 Vorbereiten wie vorhergehendes Beispiel
08 jsr 62648     ; LOADXX lädt die Datei ohne Zeiger zu ver-
  (jsr 62294)      ändern in den Arbeitsspeicher
09 jsr 63213     ; TWAIT wartet Übertragung ab
  (jsr 63787)
```

Label	C64	40/80XX
LOAD	---	62472=$f408
LOADXX	62648=$f4b8	62294=$f356
TWAIT	63213=$f6ed	63787=$f92b
LVFLAG	147=93	157=$9d

9.8 Anwendung: Lademodul zum Nachladen von Programmen

Wie schon angedeutet, hat man von BASIC-Programmen aus z.B. Maschinenmodule nachzuladen wie Spracherweiterungen, Hilfsroutinen usw.
Hier bietet sich der Einbau eines Maschinenprogramms in das Hauptprogramm an, so daß jederzeit die Möglichkeit besteht alles

Erdenkbare nachzuladen, ohne den Ablauf des Hauptprogramms abzubrechen oder mit Overlay-Techniken arbeiten zu müssen.

9.8.1 Das Maschinenprogramm "8l-loadmodul"

Beispiel:
Nehmen wir an, wir wollen zu einem Textprogramm ein schnelles Inhaltsverzeichnis der Disketten erstellen. Dieses Hilfsprogramm soll "quickdirector" heißen und auf Diskette bereitgehalten werden.
(Wir werden diesen "quickdirector" etwas später erarbeiten.)

Natürlich könnte dieses Programm auch über datas eingelesen werden, was uns natürlich zu langsam ist.
Schreiben wir also entsprechend den vorangegangenen Beispielen ein Maschinenprogramm, das dieses Laden vornimmt.

ASSEMBLER-Ladeprogramm "8l-loadmodul" (C64):

Der Kommentar ist im folgenden kurz gehalten. Sie erkennen aber doch die Struktur?

```
01 lda #0
   sta STATUS
   sta LVFLAG
02 lda #8
   sta GA        ; Gerätenummer (Floppy) nach GA
03 ldx #82       ; LO- Adresse des Programmnamens
   stx NAMADR-LO
   ldx #3        ; Hi- Adresse des Programmnamens
   stx NAMADR-HI; also <187/188>=850
04 ldx #0
   stx NAMLEN    ; Vorbereiten der Länge des Namens als Platz-
                   halter. Die zu ladende Länge wird später ein-
                   gesetzt (z.B. ldx #12).
05 jsr LOADXX    ; lädt ohne Zeigerveränderung
06 jmp TWAIT     ; wartet ab, bis zum Abschluß der
                   Eingabe, dämit nicht vorzeitig zurückgesprungen
                   werden kann.
                   Anschließend erfolgt der Rücksprung.
```

Wenn Sie dieses Programm geschrieben haben, dann steht es zum

Beispiel ab 30000 bis 30028. Sichern Sie nun diesen Bereich mit HIMEM 30000 (oder mit poke 55,48:poke 56,117 bzw. bei 40/80XX: poke52,48 und poke53,117) und springen Sie in den READY-Modus.

9.8.2 BASIC-Hilfsprogramm zur Übernahme von Maschinenteilen

Um dieses kurze, aber hilfreiche Programm von BASIC aus zu laden, schreiben wir uns eine kleine BASIC-Routine, die den Maschinencode in DATAs übernimmt, der dann vom Hauptprogramm in einen gewünschten Bereich eingelesen werden kann.

Zeilen 150-160:
Dazu lassen wir uns abfragen, welche Zeilen als DATA-Zeilen verwendet werden sollen.

Zeilen 170-180:
Dann geben wir ein, wo das Ladeprogramm beginnt (Zeile 170). In unserem Fall wäre das also 30000.
Schließlich übergeben wir noch die Adresse des letzten Bytes, in unserem Fall also 30028.

ab Zeile 190:
Im folgenden wird die Zeilennummer gedruckt und dahinter jeweils 20 Bytes sauber mit Kommata getrennt aus dem Maschinenteil ausgegeben.

Nach Beendigung des Durchlaufs lassen sich die DATA-Zeilen mit RETURN als BASIC-Zeilen übernehmen und an geeigneter Stelle im Hauptprogramm einbauen.
Unser Ladeprogramm steht nun also als DATAs in BASIC-Zeilen.

Das BASIC-Hilfsprogramm "82-masch-datas":

```
100 rem maschinenprogramm in datas
120 rem
150 input"1. DATA-BASIC-Zeile ";zl
160 input"gewünschter Zeilenabstand ";za
170 input"1. Byte in welcher Adresse ";bl
180 input"letztes Byte in Adresse    ";lb
190 printusing"#####",zl;:print" data ";
200 for i=0toll
210 print peek(bl+i)chr$(157)",";
220 if bl+i>=lb goto 300
```

```
230 next i
240 print peek(bl+i)
250 bl=bl+12:zl=zl+za:gotol90
300 end
```

Anmerkung: Wenn PRINTUSING nicht zur Verfügung steht, genügt auch das einfache PRINT.
Mit i von 0 bis 14 erfolgen 15 DATA-Drucke. Nach next i steht i auf 15. Dieses sechzehnte DATA wird als zeilenletztes ohne Komma ausgegeben, aber mit anschließendem Zeilensprung.
Wen die Lücken zwischen den Zeichen stören, kann den PRINT-Inhalt auch mit mid$(str$(peek(bl+i),2) ausgeben.

Mit diesem Programm könnten Sie auch größere Maschinenprogramme in DATAs übertragen. Aber genau das wollen wir ja mit unserem Ladeprogramm vermeiden. Eine direkte Übernahme von Diskette ist nach wie vor unser Ziel.
Das Hauptprogramm wird eigentlich nur mit den DATA-Zeilen des Ladeprogramms und ein paar wenigen Initialisierungen belastet, während eine indirekte DATA-Laderei erstens weit mehr Platz und zweitens mehr Zeit benötigt.

Nehmen wir für unsere DATAs die Zeilen 40000 und 40010, so erhalten wir mit unserem Hilfsprogramm für den C64 die Zeilen

```
40000 data 169,0,133,144,133,147,169,8,133,186,162,82,134,187
40010 data 162,3,134,188,162,5,134,183,32,184,244,32,237,246,96
```

Für den 40/80XX ergibt sich folgende Dataleiste:

```
40000 data 169,0,133,157,133,150,169,8,133,212,162,82,134,218
40010 data 162,3,134,219,162,5,134,209,32,86,243,32,43,249,96
```

Diese beiden Zeilen enthalten nun also den Maschinencode des Programms, das wir vorhin in ASSEMBLER entwickelt haben.
Mit ein paar kleinen Ergänzungen können wir es nun von BASIC aus als Modul unterbringen, wo wir wollen (bzw. wo es paßt).

Soll z.B. das Programm "lademodul" ab Adresse (30000) bis (30028) stehen und das Programm "quickdirector" laden, dann kann das so aussehen:

"83-lademodul/B":

```
40000 und 40010 wie oben
40020 '============ einlesen des moduls 'lademodul' ==============
40025 bm=30000: ' beginn des moduls
40030 for n=0 to 28
40040 read a:poke bm+n,a:nexta


40050 '==========laden des m-programms '74-seq-rout' ============
40100 dn$="74-seq-rout":lw$="0"
40110 ln=len(dn$): 'stellt die länge des dateinamens fest
40120 for k=1 to ln
40130 poke 851+k, asc(mid$(dn$,k,1))
40140 next k :' legt dateinamen zeichen für zeichen ab (852) ab
40150 poke 850, asc(lw$):poke 851, asc(":")
40151 'laufwerknummer und trennzeichen werden dem dateinamen
      vorangestellt
40180 poke bm+19,ln+2
40181 'die stringlänge steht als 19. maschinencode im modul und
      wird  jetzt  mit dem  richtigen  wert  belegt
40200 sys bm :' aufruf des lademoduls, "quickdirector" laden
```

Zu Zeile 40180:
Erinnern Sie sich: Wir haben vorhin im ASSEMBLER-Programm die Dateilänge einfach mal mit dem Wert 0 vorbesetzt. Zählen Sie nach, dann werden Sie feststellen, daß dies im Maschinencode die Stelle bm+19 ist. Mit dem POKE-Befehl bringen wir jetzt - nachdem wir die Länge des Dateinamens kennen - den richtigen Wert ein.

Sie können das Modul universell einsetzen, um beliebige Module jederzeit nachzuladen. Wenn man nämlich vom Hauptprogramm aus die Variablen dn$ und lw$ belegt, kann man sofort zu Zeile 40110 springen und das gewünschte Programm wird sofort geladen.

Selbstverständlich muß man aufpassen, daß man keine Überschneidungen zwischen Hauptprogramm, Lademodul und den nachzuladenden Modulen erzeugt.

Wird in einem Programm das Lademodul mehrfach benützt, muß es in einem durch HIMEM o.ä. geschützten Bereich untergebracht werden. (In unserem Falle wäre das HIMEM 30000.)

Benötigt man es nur am Anfang eines Programms, um zusätzliche

Hilfsroutinen zur Verfügung zu haben, genügt es, wenn es irgendwo zwischen BASIC-Ende und Stringbereich steht. Es wird dann u.U. von Variablen oder Strings überschrieben. Das macht aber nichts aus.

Übrigens: Nachdem Sie jetzt wissen, wie das Programm in BASIC aussieht, brauchen Sie sich die Mühe mit dem ASSEMBLER-Programm gar nicht mehr zu machen. Es steht Ihnen ja jetzt in den beiden DATA-Zeilen jederzeit zur Verfügung.

9.9. Modul "84-quickdirector" mit TALK, BASIN, UNTALK

Wir werden im folgenden ein etwas ausführlicheres Programm besprechen. Es ist das "quickdirector", das wir vorhin schon erwähnt haben.

Was es kann? Nun, es holt das Inhaltsverzeichnis (directory) einer Diskette und gibt es auf dem Bildschirm aus.

Nichts besonderes, werden Sie sagen, das macht der Befehl DIRECTORY aus BASIC auch.
Richtig, aber unser "quickdirector" ist - wie der Name vermuten läßt - erstens wesentlich schneller und zweitens laufen die Dateinamen nicht einfach aus dem Bildschirm nach oben hinaus, sondern werden in zwei Spalten und maximal 25 Zeilen ausgegeben, so daß bis zu 50 Dateien zugleich auf dem Schirm erscheinen können. Mehr sind in der Regel kaum auf einer Diskette. Sollte diese Zahl aber doch überschritten werden, fängt die Ausgabe wieder am oberen Bildschirmrand an, so daß immer die letzten 50 Dateien sichtbar sind.

Wir beschränken uns dabei auf den jeweiligen Namen. Um möglichst viel auf den Schirm zu bringen, rationalisieren wir weiter: Es werden keine Anführungszeichen und keine Dateitypen ausgegeben, auch die Meldung "xx blocks free" lassen wir weg.

Um die Dateinamen einwandfrei identifizieren zu können, drucken wir sie revers aus. Damit werden auch eventuelle Leerzeichen einwandfrei erkannt.

Das waren unsere Arbeitsziele. Hier der Ablaufplan (auf ein Struktogramm wird verzichtet):

Ablauf des "84-quickdirector" (Adressen C64):

Teil 1: Initialisierung

- Da nicht alle Zeichen aus dem normalen Inhaltsverzeichnis ausgegeben werden sollen, setzen wir ein Druckflag mit Adresse (1000) ein, das nur bei Belegung mit 1 Zeichenausgabe zuläßt. Initialisierung also: <1000>=0

- Wir führen zwei Schreibzeiger in SZ1=(97/98) und SZ2=(99/100) zur Steuerung der Ausgabe auf dem Bildschirm.
  Der erste Zeiger SZ1 enthält immer die Bildschirmadresse, die gerade bedruckt wird.
  SZ2 dagegen wird immer auf den Anfang des auszugebenden Namens gestellt, steht also auf dem Anfang einer Spalte in der aktuellen Zeile. Nennen wir SZ2 daher Spaltenanfangszeiger.
  Initialisierung ("HOME"): <SZ1>=<97/98>=0/4=1024
  <SZ2>=<99/100>=0/4=1024

- Adresse (1001) verwenden wir zur Kennzeichnung des Laufwerks 0 oder 1. Vorbelegung: <1001>=0

- Das Inhaltsverzeichnis wird intern unter dem Namen "$" geführt. "$0" steht auf Laufwerk 0, "$1" entsprechend auf 1.
  Die Adressen (850/851) verwenden wir zur Aufnahme einer dieser beiden Dateinamen.
  Bei der Singlefloppy 1541 lautet der Name nur "$", ist also einstellig.

- Ansonsten wird wie üblich belegt:
  z.B. logische Filenummer <LA> = 14
  Gerätenummer <GA> = 8
  Sekundäradresse <SA> = 96
  Zeiger auf Namen <NAMADR> = 850
  Länge des Namens <NAMLEN> = 1

Teil 2: Floppy zur Ausgabe vorbereiten

- Nach der Initialisierung kann die Datei "$" bzw. "$0" geöffnet werden.
  Bei entsprechender Änderung läßt sich dieses Programm natürlich auch verwenden, um andere Dateien auszugeben (dazu Namen, Länge usw. austauschen).

- Nach dem Zurücksetzen des Status-Bytes auf 0, wird das Floppy als "Talker" (Sender) aktiviert mit **TALK.**

Teil 3: Daten holen und verarbeiten

- Holen eines Bytes vom (IEC-)Bus mit **BASIN**

- Nachdem wir uns nur für die Tastaturzeichen interessieren, (Code 1 bis 63), subtrahieren wir von allen Zeichen mit höherer Codenummer den Wert 64.
  (Das läßt sich auch mit AND #63 erledigen.)

- Eine Ausgabe erfolgt solange nicht, bis wir auf das erste Anführungszeichen stoßen.
  Jetzt kann das Druckflag endlich mal auf 1 gesetzt werden und alle folgenden Zeichen werden ausgegeben, solange bis wieder das nächste Anführungszeichen (Code 34) auftaucht.

- Beim zweiten Anführungszeichen wird das Druckflag wieder ausgeschaltet und die nächste Spalte vorbereitet, also der Spaltenanfangs-Zeiger in **SZ2** um 16 erhöht.

Teil 4: Schreibzeiger wieder auf Bildschirmanfang setzen

- Wird die untere Zeile des Bildschirms erreicht, wird zunächst geprüft, ob auch schon die zweite Spalte erreicht ist.
  Wenn ja, wird der Schreibanfangs-Zeiger wieder auf links oben eingestellt.

- In jedem Fall wird der Schreibanfangs-Zeiger nach **SZ1** als Basis für die Druck-Routine übertragen.

Teil 5: Löschen eventueller Voreinträge

- Um eventuelle Einträge ohne Reste zu überschreiben, löschen wir nach jeder fertig geschriebenen Datei, die nächsten 20 Zeichen auf dem Schirm.

Teil 6: Ausgabe

- Die eigentliche Druckroutine erzeugt zunächst einmal von jedem Zeichen den entsprechenden REVERS-Code, gibt das Zeichen auf dem Schirm aus und erhöht den Schreibzeiger um 1.

Teil 7: Bildschirm-Halt

- Damit auch längere Inhaltsverzeichnisse nicht über den Bildschirm hinauslaufen können, bevor man sie richtig angeschaut hat, legen wir die Leertaste (Tastaturcode 60 beim C64) als

Haltetaste und die Taste "Cursor rechts" (Code 2) als Abbruchmöglichkeit fest.

Teil 8: Abschluß

- Zum Schluß wird geprüft, ob der Status noch Null ist. Wenn nicht, dann liegt ein Fehler vor, was eigentlich nur heißen kann: Datei, also Inhaltsverzeichnis ist zu Ende.

- Der Abschluß erfolgt dann, indem das Floppy wieder in den Ruhestand geschickt und die Datei geschlossen wird.

Anmerkung:
Man kann auch mit einem Schreibzeiger, z.B. **SZ1** auskommen, wenn man beim Drucken den y-Index mitlaufen läßt.

Auf einem 80-Zeichen-Schirm geht es natürlich großzügiger zu: Man hat genügend Platz, um die Dateitypen mit ausgeben zu lassen oder die Spaltenzahl anders zu wählen: Bei vier Spalten hätten wir z.B. 20 Zeichen Platz für jede Datei und brauchten nicht so sehr mit den Abständen zu geizen. Richten Sie sich also Ihren "quickdirector" her, wie Sie ihn gern hätten. Das folgende Assemblerprogramm soll nur Anregung sein.

Siehe dazu auch Abschnitt 7.7!

Das ASSEMBLER-Programm "84-quickdirector" (C64):

Teil 1: Initialisieren

```
- 10000 ldx #0
  10002 stx 1000     ; Druckflag "aus"
- 10005 stx 97       ; Bildschirmanfang LO von 1024
  10007 stx 99       ; Schreibzeiger dto.
  10009 ldx #4       ; dto. HI von 1024
  10011 stx 98       ; als Bildschirmanfang
  10013 stx 100      ; und Schreibzeiger
- 10015 ldx #36      ; Code für "$"(Dateiname)
  10017 stx 850      ; ab (850) ablegen
( 10020 ldx 1001     ; Laufwerknummer: <1001>=0 oder 1) nicht C64
( 10023 stx 851      ; hinter den Namen setzen)         nicht C64
...     nop
- 10027 lda #14      ; log. Dateinummer z.B.14
  10029 sta 184      ; in LA bereitstellen
  10031 lda #8       ; Geräteadresse 8 (Floppy)
```

```
  10033 sta 186      ; in GA bereitstellen
  10035 lda #96      ; Sekundäradresse mit gesetzten Bits 5 und 6
                       (oder Sekundäradresse 0)
  10037 sta 185      ; in SA bereitstellen
  10039 lda #82      ; LO der Anfangsadresse des Namens (ab 850)
  10041 sta 187      ; nach NAMADR-LO und
  10043 lda #3       ; HI nach
  10045 sta 188      ; NAMADR-HI: (3 mal 256 + 82 = 850)
  10047 lda #1       ; Länge des Namens (z.B. von "$")
  10049 sta 183      ; nach NAMLEN

Teil 2: Floppy aktivieren
  10051 jsr 62282    ; OPEN öffnet die Datei z.B. "$"
  10054 lda #0       ; Null in das ...
  10056 sta 144      ; Status-Byte bringen
   ...  nop ...

Teil 3: Datenverarbeitung
- 10061 ldx #14      ; Filenummer laden
  10063 jsr 65478    ; und mit CHKIN Eingabe vorbereiten
- 10066 jsr 65487    ; BASIN holt ein Byte aus der Datei
  10069 cmp #64      ; mit 64 vergleichen
  10071 bmi 10078    ; kleiner 64 ===> Sprung nach (10078)
  10073 sec          ; sonst 64
  10074 sbc #64      ; subtrahieren
...     nop
- 10078 ldy #0       ; Druckflag
  10080 cpy 1000     ; ein?
  10083 beq 10143    ; nein ===> weiteres Byte holen
  10085 cmp #34      ; ja ===> mit Code für Gänsefuß vergleichen
  10087 bne 10165    ; kein Gänsefuß  ===> Sprung zu Teil 6
  10089 ldy #0       ; Gänsefuß ===> Druckflag
  10091 sty 1000     ; "aus" (Dateiname zu Ende)
- 10094 lda 99       ; Spaltenanfangszeiger LO
  10096 clc
  10097 adc #20      ; um 20 erhöhen (bei 2-spaltiger Ausgabe)
  10099 sta 99       ; und ablegen
  10101 bcc 10105    ; kein Überlauf ===> weiter
  10103 inc 100      ; sonst Spaltenanfangszeiger HI erhöhen

Teil 4: Spaltenanfangszeiger zurück (falls erforderlich):

- 10105 lda 100     ; HI des aktuellen Namensanfangs
  10107 cmp #7      ; mit unterer Grenze vergleichen
```

```
  10109 bmi 10133   ; noch nicht erreicht ===> weiter
  10111 lda 99      ; LO des aktuellen Namensanfangs
  10113 cmp #127    ; letzte Spalte erreicht?
  10115 bmi 10133   ; nein ===> weiter
  10117 lda #0      ; ja ---> Zeiger auf Bilschirmanfang
  10119 sta 97
  10121 sta 99        (wie Teil 1)
  10123 lda #4
  10125 sta 98
  10127 sta 100
  10129 bne 10143   ; unbedingter Sprung
...     nop
- 10133 lda 99      ; Schreibzeiger auf Spaltenanfang
  10135 sta 97      ; LO
  10137 lda 100     ; und
  10139 sta 98      ; HI setzen
  10141 bne 10175   ; unbedingter Sprung, <A> wird nie Null
```

Teil 5: Platz schaffen vor Ausdruck eines neuen Namens

```
- 10143 cmp #34     ; letztes Zeichen Gänsefuß, also Namensende?
  10145 bne 10175   ; nein ===> weiter
  10147 ldx #1      ; ja ---> Druckflag
  10149 stx 1000    ; auf "ein" schalten
- 10152 ldy #19     ; Zähler (X) auf 19
  10154 lda #32     ; Code für Leerzeichen
  10156 sta (99),y  ; 19 mal  ausgeben (Spaltenbreite zum even-
                      tuellen Überschreiben freimachen)
  10158 dey         ; Zähler auf Null?
  10159 bne 10156   ; nein ===> weitermachen
  10161 clc
  10162 bcc 10175   ; und unbedingter Sprung ans Ende
        nop
```

Teil 6: Zeichenausgabe

```
- 10165 ora #128    ; Reversbit setzen
  10167 sta (97),y  ; und Zeichen auf den Bildschirm
  10169 inc 97      ; Schreibzeiger um 1 erhöhen
  10171 bne 10175   ; kein Überlauf ===> Sprung
  10173 inc 98      ; sonst HI des Schreibzeiger auch +1
```

Teil 7: Bildschirm halt

```
- 10175 lda 203     ; KEY abfragen
```

```
  10177 cmp#2        ; "Cursor rechts"?
  10179 beq 10189    ; ja ===> Ende
- 10181 cmp #60      ; Leertaste ?
  10183 beq 10175    ; ===> KEY neue abfragen (warten)
```

Teil 8: Abschluß

```
  10185 lda 144      ; STATUS prüfen
  10187 beq 10066    ; gleich 0 ===> weiteres Byte holen
- 10189 jsr 65484    ; sonst mit CLRCH Floppy deaktivieren
  10192 lda #14      ; LA laden
  10194 jsr 62097    ; und mit CLOSEA Datei schließen
  10197 rts          ; Rücksprung zum aufrufenden Programm
```

Nach dem Aufruf mit SYS 10000 oder einem entsprechenden Befehl, erscheint ein 2-spaltiges Inhaltsverzeichnis von Drive 0. Wenn man sich die Möglichkeit für beide Laufwerke vorbehalten will, belegt man die Adressen (10002) bis (10004) mit NOP und setzt in (1000) die Nummer für das gewünschte Laufwerk ein.

Abänderungen für den 40/80XX:

- SZ1 und SZ2 müssen in freien Zeropage-Adressen liegen. Dafür bietet sich ebenfalls ein FAC an:
- SZ1=(94/95) und SZ2=(96/97)
- **NAMLEN** ist 2 Zeichen lang ("$1" oder "$0")
- statt STATUS,TALK und SASENT kann man ab (10054) programmieren: ldx #14 und jsr **CHKIN**
- Beim 80-Zeichenschirm ist eine bis zu fünfspaltige Ausgabe des Inhaltsverzeichnisses sinnvoll. Deshalb erhöht sich der Spaltenzeiger immer um 16 Adressen (10097).
- Um freien Platz zu schaffen, muß der Zähler bei 10152 auf 15 gesetzt werden.
- Statt **CLRCH** kann auch **UNTALK** verwendet werden.
- Zum Schließen der Datei kann **CLOSEL** mit der laufenden logischen Adresse **LA** verwendet werden.

9.10 Modul "85-printdirector" (Floppy ---> Drucker)

Wie man ROM- Routinen sinnvoll einsetzen kann, sehen wir am folgenden Beispiel. Es ist noch etwas umfangreicher als das vorhergehende. Aber schließlich haben wir ja schon Fortschritte beim Analysieren und Strukturieren gemacht, so daß auch dieser "Fall"

nicht unlösbar sein dürfte.
Wenn Sie den "84-quickdirector" durchgearbeitet und zum fehlerfreien Laufen gebracht haben, sollten Sie den folgenden Teil eigentlich (fast) problemlos mitverfolgen können.

Zunächst unsere Zielangabe:

Wir entwerfen ein ASSEMBLER-Programm, mit dem man das Inhaltsverzeichnis einer Diskette auf den Drucker ausgeben kann und zwar dreispaltig, mit dem Diskettennamen als Kopf, ohne die Angaben "Zahl der Blöcke" und "xxxx blocks free", aber mit der jeweiligen Dateiart (prg, rel, seq, usr).
Als Drucker nehmen wir ein EPSON-Gerät an, das mit Gerätenummer 4 und Sekundäradresse 8 angesprochen werden kann. Dabei lassen wir die Möglichkeit offen, daß der Drucker am USER-Port hängt. In letzterem Fall kann es notwendig werden, die Zeropage-Adresse **DEVOUT** mit 4 zu belegen, um den Drucker als aktives Ausgabegerät zu bestimmen. Hier müßten Sie das Programm Ihren Verhältnissen anpassen.

Anmerkung für die angehenden Profis:
Wie in der Einleitung schon erwähnt, haben wir bisher nicht allzu großen Wert auf Sparsamkeit und Raffinesse bei der Erstellung unserer Programme gelegt. Nun wollen wir aber doch einmal ein paar Feinheiten einbauen.
Da wären z.B. der Einsatz einer Byte-Leiste, aus der die Druckersteuerungsdaten übernommen werden und auch der Kniff mit dem Zwischensprung, der verhindert, daß wir auf den JMP-Befehl zurückgreifen müssen und damit unser Prinzip des frei verschiebbaren Moduls nicht beibehalten können.
Schauen Sie sich deshalb den Ablauf und das ASSEMBLER-Programm so lange an, bis Sie jeden Schritt verstanden haben.

Ablauf (Grobstruktur) des "85-printdirector":

Teil 1: Initialisieren

- Drei Adressen im Kassettenpuffer werden als Flags bzw. Zähler verwendet und erhalten einen Ausgangswert:

  <900>=0 als Druckflag, das erst gesetzt wird, wenn das erste Anführungszeichen gefunden wurde.

<1017>=1 Flag für den Druck des Kopfes, das zurückgesetzt wird, sobald der Kopf (Diskettenname) gedruckt wurde.

<1018>=1 Zähler für die Anzahl der Spalten, der zunächst auf 1 steht, und dann auf 3 gesetzt wird.

<1003>=49 Code für die Laufwerknummer, die auch von einem aufrufenden Programm aus gesetzt werden kann (nicht für C64).

- Datei auf die Floppy öffnen:
  Dateiname FILNAM= "$" (40/80XX: "$0" oder "$1")
  abgelegt ab NAMADR (Adresse des Dateinamens) LO/HI=82/3=850
  mit der Länge <NAMLEN>=2
  logische Adresse <LA>=14
  Gerätenummer <GA>=8
  Sekundäradresse <SA>=96 (oder 0)

- Drucker öffnen:
  Je nach Druckertyp und Druckmodus sind die Adressen zu wählen.

  z.B.: kein Name notwendig
  Geräteadresse <GA>=4
  logische Adresse <LA>=8
  Sekundäradresse <SA>=8

  Wenn der Drucker über den USER-Port angeschlossen ist oder wenn Sie keine Sekundäradresse benötigen, können Sie auch die Gerätenummer in die Zeropageadresse **DEVOUT** setzen und **SA** mit 255 belegen.

  DEVOUT ist (154)/C64 bzw. (176)/40/80XX und enthält die Nummer des aktiven Ausgabegeräts.

  Sie brauchen jetzt nur noch **LISTEN** aufrufen und eventuell die Sekundäradresse mit **SASEN**L ausgeben. Ein OPEN erübrigt sich dann. In unserem Beispiel für den C64 haben wir diese Version gewählt.

  An dieser Stelle müssen Sie also das Programm Ihren eigenen Erfordernissen anpassen.

- Drucker initialisieren
  Da der Drucker dreispaltig arbeiten soll, müssen wir drei Tabulatorstops programmieren.

Anschließend wird ein Zeilenvorschub ausgegeben und der erste TAB-Stop angesprungen.

In BASIC sieht das so aus:

```
print#8,chr$(27)"d"chr$(5)chr$(30)chr$(55)chr$(0);
print#8,chr$(10)chr$(9);
```

Der Drucker erwartet über die logische Adresse 8 also die Bytefolge 27,68,5,30,55,0 und 10,9.

Diese Bytefolge legen wir in einer Leiste ab, die von (10107) bis (10114) geht, also die genannten acht Bytes umfaßt.
Mit einer Schleife geben wir diese Steuerdaten auf den Drucker aus.

Teil 2: Unterprogramm 'Eingabe'

- Das Inhaltsverzeichnis ist immer so angelegt, daß 32 Bytes zu einem Namen gehören.

  Ab dem 4. Byte steht der Name in Anführungszeichen, dahinter nach Leerzeichen (zum Auffüllen) die Art der Datei (z.B. 'prg' oder 'seq').

  Wir laden daher das Inhaltsverzeichnis 32-byte-weise und legen so einen Namensblock erst einmal ab (5000) an (Beispiel!).

Teil 3: Abschluß der Ein/Ausgaben

- Die Statusvariable wird mit 64 belegt, wenn die Eingaberoutine auf den Schluß (3 mal Code 0) stößt.

  Ist dies der Fall, dann werden die beiden Dateien für Floppy und Drucker geschlossen. Danach erfolgt der Rücksprung zur aufrufenden Routine.

Teil 4: Unterprogramm 'Ausgabe'

- Drucker als Empfänger (Listener) schalten.
  Dazu setzen wir in die Zeropage-Adresse **DEVOUT** die Geräteadresse des Druckers.

- Die ersten Zeichen des Namensblocks auf Anführungszeichen hin (Code 34) absuchen. Wenn gefunden, Druckflag <900> = 1 setzen und alle Zeichen ausgeben bis einschließlich Dateityp.

- Nach der ersten Zeile (Diskettenname) einen Zeilenvorschub aus-

geben, den ersten TAB-Stop anspringen und

- den Zähler für die Spaltenanzahl immer dann auf 3 stellen, wenn eine neue Zeile angefangen werden muß.
  Außerdem Ausgabe eines Zeilenvorschubs und eines TABs.

- Drucker deaktivieren, Druckflag wieder auf Null setzen und über eine Zwischenstation wieder an den Anfang der Schleife (zur Eingabe-Routine) springen.
  Der Zwischensprung ist notwendig, weil mit den Branchbefehlen eben nur maximal 128 Adressen weit gesprungen werden kann.

Das ASSEMBLER-Programm "85-printdirector" (C64):

Teil 1: Initialisieren

```
-       nop
  10005 ldx #0       ; Null als Flag '
  10007 stx 900      ; für Druck ein/aus
  10010 ldx #1       ; 1 als Flag
  10012 stx 1017     ; für Drucken des Kopfes (Diskettenname)
  10015 stx 1018     ; als Spaltenzähler
        nop
- 10020 lda #36      ; Code für '$'
  10022 sta 850      ; nach (850) als erstes Namensbyte
        nop

  10032 lda #14      ; logische Adresse
  10034 sta 184      ; nach LA
  10036 lda #8       ; Gerätenummer (Floppy)
  10038 sta 186      ; nach GA
  10040 lda #96      ; Sekundäradresse
  10042 sta 185      ; nach SA
  10044 lda #82      ; LO-Byte der Anfangsadresse (850)
  10046 sta 187      ; nach NAMADR-LO
  10048 lda #3       ; HI-Byte der Anfangsadresse (850)
  10050 sta 188      ; nach NAMADR-HI
  10052 lda #1       ; Länge des Namens
  10054 sta 183      ; nach NAMLEN
  10056 jsr 62282    ; und mit OPEN Datei auf das Floppy öffnen
        nop
- 10060 lda #4       ; Geräteadresse 4
  10062 sta 154      ; nach DEVOUT (falls erforderlich)
  10064 sta 186      ; und GA
  10066 lda #8       ; Sekundäradresse
```

```
  10068 sta 185     ; bereitstellen in SA
 (10070 sta 184     ; evtl. logische Adresse bereitstellen
  10072 jsr 62282   ; und mit OPEN Datei auf den Drucker öffnen
                      nicht bei Ansprechen mit DEVOUT)
        nop
- 10075 lda #0      ; Null
  10077 sta 144     ; nach STATUS
  10079 jsr 60684   ; und Drucker mit LISTEN aktivieren
  10082 lda 185     ; Sekundäradresse mit
  10084 jsr 60857   ; SASENL ausgeben (Kanal bereitstellen)

- 10087 ldx #8      ; (X) als Zähler
  10089 lda 10106,x ; und die ab (10107) stehenden acht
  10092 jsr 65490   ; Bytes mit BSOUT
  10095 dex         ; ausgeben
  10096 bne 10089   ; Schleifenende erreicht?
  10098 nop
- 10099 jsr 63060   ; ja ===> Drucker mit UNLISN deaktivieren
  10102 clc         ; und die nun folgende Byteleiste
  10103 bcc 10116   ; auf jeden Fall überspringen
        nop
- <10107> = 9       ; Code für TAB
  <10108> = 10      ; Code für Linefeed
  <10109> = 0       ; Begrenzung für Tabulatorsetzen
  <10110> = 55      ; 3. TAB
  <10111> = 30      ; 2. TAB
  <10112> = 5       ; 1. TAB
  <10113> = 68      ; Code für "d" (Epson-System für TABSET)
  <10114> = 27      ; ESC-Code
        nop
```

Teil 2: Unterprogramm 'Einlesen der Daten'

```
- 10116 lda #14     ; mit logischer Adresse und den Routinen
  10118 jsr 62223   ; SUFTAB und
  10121 jsr 62239   ; SETTAB die Floppydatei ansprechen
  10124 ldx #14     ; logische Adresse der Floppydatei laden
  10126 jsr 65478   ; und mit CHKIN Eingabe vorbereiten
        nop
- 10136 ldx #32     ; und 32 Bytes aus der Datei
  10138 jsr 65487   ; einzeln mit BASIN holen
  10141 sta 5000,x  ; und z.B. von (5001) bis (5032) ablegen
  10144 lda 144     ; STATUS prüfen: Ende erreicht?
  10146 bne 10157   ; ja ===> zum Teil 3 springen
  10148 dex         ; sonst Schleife weiter abarbeiten
```

```
  10149 bne 10138   ; bis Zähler auf Null ist
  10151 jsr 65484   ; mit CLRCH Standard I/O setzen
  10154 clc
  10155 bcc 10173   ; und Teil 3 überspringen

Teil 3: Abschlußroutine
- 10157 lda #14     ; Nummer des Floppy-Files
  10159 jsr 62097   ; CLOSEA schließt Floppy-Datei
  10162 jsr 65511   ; CLALL schließt den Rest
  10165 rts
     .. nop ..
Zwischenstation für den Sprungbefehl:
  10170 clc         ; unbedingter Sprung
  10171 bcc 10116   ; zur Eingaberoutine

Teil 4: Unterprogramm 'Drucken' (Ausgabe):
- 10173 lda #4      ; mit Gerätenummer 4
  10175 sta 154     ; DEVOUT als aktives Ausgabegerät
  10177 sta 186     ; sowie GA belegen
  10184 lda #255    ; für unbenötigte Sekundäradresse 255
  10181 sta 185     ; nach SA
  10183 lda #0      ; Null
  10185 sta 144     ; nach STATUS
  10187 jsr 60684   ; mit LISTEN Drucker aktivieren
  10190 lda 185     ; Sekundäradresse ausgeben
  10192 jsr 60857   ; mit SASENL
        nop
- 10196 ldx #32     ; Zähler, um Zeichen aus dem Block zu holen
  10198 lda 5000,x  ;
  10201 cmp #34     ; Anführungszeichen gefunden?
  10203 bne 10210   ; nein ---> überspringen der Flagsetzung
  10205 ldy #1      ; ja ---> Druckflag mit 1
  10207 sty 900     ; belegen
  10210 cpx #3      ; 3 Zeichen vor Blockende wird aufgehört
  10212 beq 10226   ; zu drucken
- 10214 ldy 900     ; Druckflag gesetzt?
  10217 beq 10223   ; nein ===> Druckerausgabe überspringen
  10219 nop
  10220 jsr 65490   ; ansonsten Ausgabe mit BSOUT
  10223 dex         ; Zähler erniedrigen
  10224 bne 10198   ; und an den Schleifenanfang springen
- 10226 ldy 1017    ; Flag für erste Zeile (Kopf) noch gesetzt?
  10229 beq 10241   ; nein ===> Linefeed usw. überspringen
  10231 ldy #0      ; ja ---> Flag zurücksetzen
```

```
  10233 sty 1017
  10236 lda #10      ; Code für Linefeed
  10238 jsr 65490    ; ausgeben mit BSOUT
- 10241 dec 1018     ; Spaltenzähler um eins heruntersetzen
  10244 bne 10256    ; Spalte gleich 0, also Zeile voll?
  10246 ldy #3       ; ja ---> Spalte wieder mit 3 vorbelegen
  10248 sty 1018     ; in Spaltenzähler
  10251 lda #10      ; und ein Linefeed
  10253 jsr 65490    ; mit BSOUT ausgeben
  10256 lda #137     ; Code für TAB-Sprung
  10258 jsr 65490    ; ausgeben (nächste TAB-Position)
- 10261 jsr 60926    ; mit UNLISN Drucker deaktivieren
  10264 ldy #0       ; Druckflag
  10266 sty 900      ; auf "nicht drucken" setzen
  10269 clc          ; und unbedingter
  10270 bcc 10170    ; Sprung zunächst nach 10170, von wo aus ein
                       Weitersprung bis zur Routine für das Ein-
                       lesen des nächsten Namensblocks erfolgt
```

Dieses Programm kann nun mit SYS 10005 aufgerufen oder von einem Maschinenprogramm mit JSR 10005 angesprungen werden. Es druckt dann das Inhaltsverzeichnis der Diskette, die gerade im Laufwerk liegt, dreispaltig aus.

Wird dieses Modul in einen anderen Bereich verschoben, muß der Einsprung eben entsprechend mitgerückt werden.

Zu beachten ist hier noch, daß auch die Ladeadresse für die Byte-Leiste (bei 10089) nachgestellt werden muß, wenn dieses Programm in einem anderen Bereich liegt. Wer das vermeiden möchte, der muß die Druckerausgaben mit einer LDA/STA- Folge bewerkstelligen. Der Platzbedarf ist dann etwas größer, aber dafür ist uneingeschränkte Verschiebbarkeit gegeben.

Zur besseren Übersicht und für eventuelle Erweiterungen, z.B für den Einsatz eines Doppelfloppys, aber auch für eventuelle Umstellungen bei Verwendung eines anderen Druckers o.ä. haben wir eine ganze Reihe von NOPs gesetzt, die zum Teil über mehrere Adressen gehen. Wenn Sie das Programm so übernehmen wollen, wie es hier steht, dann füllen Sie bitte diese Lücken vollständig mit dem ASSEMBLER-Befehl NOP auf.

Läuft Ihr Programm auf diese Weise einwandfrei, dann können Sie alles handlich zu einem kompakten Dienstprogramm zusammenschie-

ben. In Sekundenschnelle erhalten Sie mit diesem Modul einen Ausdruck Ihres Inhaltsverzeichnis, der Ihnen mit Sicherheit dazu verhilft, die Übersicht über Ihre Diskettenaufzeichnungen zu verbessern.

9.11 Direktzugriffe auf Floppy - Modul "86-fastdisk" (8050)

Zum Abschluß der (IEC-)Bus-Operationen schauen wir uns noch ein Beispiel für die Direktzugriffsbefehle auf die Diskettenstation an. Alle anderen Direktzugriffe werden analog ausgeführt. (Siehe dazu das jeweilige Floppy-Handbuch!)

Diesmal sind vorwiegend die 40/80XX-Besitzer angeprochen, die in den meisten Fällen mit der 8050-FLoppy oder deren Nachfolgern arbeiten:

Untersucht man das DOS-Listing für die 8050-Diskettenstation, so findet man in den Adressen (4096) bis (4098) Steuerdaten, die die Geschwindigkeit des Geräts beeinflussen:

<4096> gibt die Zeitintervalle zwischen den einzelnen Interrupts an und steht normalerweise auf 7 (=0.77 Millisekunden).
<4097> steuert die Länge der Motoranlaufzeit und ist mit 14 belegt, was 1,54 Sekunden bedeutet.
<4098> legt die Motornachlaufzeit fest mit 45, also ca. 5 Sekunden.

Diese Werte lassen sich nun um einiges verkürzen, so daß Sie Ihre 8050-Station auf "fast" laufen lassen können, ohne wesentlich an Übertragungssicherheit einzubüßen.

Es empfiehlt sich aber, folgende Werte nicht zu unterschreiten, da sonst Probleme auftreten wie ständiges Ein- oder Ausschalten des Motors bei Ladevorgängen o.ä.:

<4096> = 5 ; <4097> = 1 ; <4098> = 60

Die Nachlaufzeit muß deswegen länger werden, damit sie sich mit dem Anlaufzeitpunkt überschneidet. Sonst geht die An- und Ausschalterei los.

In BASIC läßt sich dies folgendermaßen realisieren:

```
100 open1,8,15 : rem befehlskanal 15
110 print#1,"m-w"chr$(0)chr$(16)chr$(3)chr$(6)chr$(1)chr$(64);
120 close1
```

Es werden also 3 Bytes ab Speicherstelle 16/0 (=4096) in das RAM der Floppy übertragen.
Das entspricht der Bytefolge 77,45,87,0,16,3,6,1,64.

Zur Übertragung in Maschinensprache braucht bei Direktzugriffen das Gerät nur als LISTENER aktiviert werden, ein File ist nicht notwendig. Der Abschluß erfolgt durch **UNLISN.**

Es werden also über den Befehlskanal 15 direkt Daten in den RAM-Bereich des DOS geschrieben (hier in den Steuerdatenpuffer).

Auf diese Weise lassen sich auch die anderen Puffer des DOS-RAMs beschreiben (z.B. mit Maschinenroutinen).

ASSEMBLER-Programm für "86-fastdisk" (8OXX/8050):

```
01 20000 lda #8        ; Gerätenummer
   20002 sta 212       ; nach GA
   20004 jsr 61653     ; LISTEN aktiviert Floppy als LISTENER
   20007 lda #111      ; Befehlskanal 15 + 96 als Sekundäradresse
   20009 jsr 61763     ; SASEND sendet Sekundäradresse usw.

02 20012 lda #77       ; Code für "m"
   20014 jsr 61854     ; OUTBUS gibt "m" aus
   20017 lda #45       ; Code für "-"
   20019 jsr 61854     ; OUTBUS
   20022 lda #87       ; Code für "w"
   20024 jsr 61854
   20027 lda #0        ; LO der Speicheradresse (4096)
   20029 jsr 61854
   20032 lda #16       ; HI von (4096)
   20034 jsr 61854
   20037 lda #3        ; Anzahl der folgenden Bytes
   20039 jsr 61854
   20042 lda #6        ; Intervall-Byte
   20044 jsr 61854
   20047 lda #1        ; Motoranlauf-Byte
   20049 jsr 61854
   20052 lda #64       ; Motornachlauf-Byte
```

```
   20054 jsr 61854     ; OUTBUS

03 20057 jmp 61881     ; UNLISN deaktiviert Bus
```

Man kann natürlich die 9 Bytes auch mit Hilfe einer angehängten Byte-Leiste und einer Zählschleife ausgeben. Aber dann verliert man die Verschiebe-Eigenschaft des Moduls, wenn man die Startadresse indiziert. (Es lassen sich bloß ca. 20 Bytes sparen.)

Gibt man dieses "fastdisk" mit SYS 20000 aus, so bleibt diese Schnellauf-Funktion bis zum RESET der Floppy erhalten. Man merkt die Wirkung sofort: Beim Speichern oder Laden ist das Gerät im Nu "da".

Adressen und ROM-Routinen zur Ein/Ausgabe

Label	C64	40/80XX
LA	184=$b8	210=$d2
	logische Adresse zur Bezeichnung der Datei	
SA	185=$b9	211=$d3
	Sekundäradresse zur Kanalbereitstellung	
GA	186=$ba	212=$d4
	Gerätenummer für angesprochenes Gerät	
DEVIN	153=$99	175=$af
	Geräteadresse GA des aktiven Eingabegeräts	
DEVOUT	154=$9a	176=$b0
	Geräteadresse des aktiven Ausgabegeräts	
NAMADR	187/188=$bb/bc	218/219=$da/db
	LO/HI der Anfangsadresse des Dateinamens oder eines Befehlsstrings	
NAMLEN	183=$b7	209=$d1
	Adresse zur Bereitstellung der Länge des Namens	

STATUS	144=$90	150=$96
	Byte für Fehler- bzw. Ende-Erkennung der Datei kein Fehler: <STATUS>=0	
LVFLAG	147=$93	157=$9d
	Load- bzw. Verify-Flag. Laden: <LVFLAG>=0	

Label	C64	40/80XX
OPEN	62282=$f34a	62819=$f563
	öffnet eine Datei mit LA,GA,SA auf ein Gerät bei Floppy-Dateien sind NAMAD und NAMLEN notwendig	
BFOUT	---	55963=$da9b
	gibt ähnlich wie OPEN einen Befehlsstring aus zugeordnet **LA, GA, SA**	
LOAD	---	62472=$f408
	lädt Datei oder Programm, das mit OPEN geöffnet wurde, dazu vorher <STATUS>=0, <LOVE>=0 setzen Programmzeiger 'Anfang' und 'Ende' werden gesetzt	
LOADXX	62648=$f4b8	62294=$f356
	wie LOAD, aber ohne Veränderung der Zeiger	
SUFTAB	62223=$f30f	62145=$f2c1
	stellt für eine bereits geöffnete Datei GN,SA,LA bereit aus der intern geführten Tabelle Vorbereitung: <LA> ---> (A)	
SETTAB	62239=$f31f	62157=$f2cd
	setzt die mit SUFTAB gefundenen Parameter in die vorgesehenen Zeropage-Adressen	
CLOSEA	62097=$f291	62178=$f2e2
	schließt eine noch offene Datei Vorbereitung: logische Adresse nach (LA) bringen	
TWAIT	63213=$f6ed	63787=$f92b
	verhindert vorzeitigen Rücksprung, wenn die Zentraleinheit schneller als das Peripheriegerät ist (z.B. beim Laden eines Programmes notwendig)	

TALK 60681=$ed09 61650=$f0d2
aktiviert das mit OPEN angesprochene Gerät als Sender (Talker)
Vorbereitung: <STATUS>=0

UNTALK 60911=$edef 61878=$f1b6
versetzt den (IEC-)Bus nach TALK wieder in den neutralen Zustand, Gerät wird als Talker deaktiviert

LISTEN 60684=$ed0c 61653=$f0d5
aktiviert das angesprochene Gerät als Empfänger
Vorbereitung: OPEN, <STATUS>=0

UNLISN 60926=$edfe 61881=$f1b9
versetzt den (IEC-)Bus nach Tätigkeit als Listener wieder in neutralen Zustand

GETSA ----- 55599=$d92f
nächste freie Sekundäradresse holen ---> SA

SASENL 60857=$edb9 61763=$f143
SASENT 60871=$edc7 61763=$f143
sendet die Sekundäradresse und bereitet damit einen Kanal für die folgende Ein- oder Ausgabe vor.
Dieser Befehl muß also immer dem ersten INBUS oder dem ersten OUTBUS vorangehen.
Vorbereitung: <A>=<SA>, also Akku mit Sekundäradresse belegen, die man eventuell aus (SA) holt

BSOUT(CHROUT) 65490=$ffd2 KERNAL 65490=$ffd2
OUTBUS 60893=$eddd 61854=$f19e
sendet das im (A)-Register befindliche Byte über den IEC-Bus auf den aktiven Kanal
Vorbereitung: OPEN, LISTEN, SASEND, <A>

BASIN(CHRIN) 65487=$ffcf KERNAL 65487=$ffcf
holt ein Zeichen nach (A) über aktiven Kanal.
Bei Standard-I/O: mit Bildschirmausgabe bis CR anschließend erstes Zeichen in (A).

INBUS 60947=$ee13 61888=$f1c0
holt über den (IEC-)Bus ein Byte nach (A) vom aktiven Kanal
Vorbereitung: OPEN, TALK, SASEND

CHKOUT 65481=$ffc9 KERNAL 65481=$ffc9
leitet Ausgabe über den (IEC-)Bus auf das angesprochene Gerät um (statt Schirm)
Vorbereitung: OPEN, <LA> ---> (X)

CHKIN 65478=$ffc6 KERNAL 65478=$ffc6
Eingabe über den (IEC-)Bus vom angesprochenen Gerät (statt Tastatur)
Vorbereitung: OPEN, <X>=<LA>

CLRCH 65484=$ffcc KERNAL 65484=$ffcc
schaltet als Eingabegerät die Tastatur (GA=0) und als Ausgabegerät den Bildschirm (GA=3)
schließt alle Kanäle, aber keine Dateien

CLALL 65511=$ffe7 KERNAL 65511=$ffe7
schließt alle Kanäle und Dateien

Der (IEC-)Bus befindet sich in einem der Zustände LISTEN, TALK oder Wartestellung (neutral).

10

Maschinenmodule in BASIC-Programmen

10 Maschinenmodule in BASIC-Programmen

10.1 Übernahme von BASIC-Parametern

Will man von BASIC aus ein Maschinenprogramm aufrufen, dann geschieht das in der Regel mit dem Befehl SYS xxxx. Werden zur Durchführung dieses "SYS-Programmes" keine festen Werte verarbeitet, sondern ergeben sich diese erst aus dem Verlauf des BASIC-Programmes, dann müssen diese sog. **Parameter** auf irgendeine Weise vom Maschinenprogramm übernommen werden können.

Zur Verdeutlichung ein Beispiel:
Wir haben im Kapitel 7 eine Routine kennengelernt, die den Bildschirm-Cursor beliebig positionieren kann. Im CBM-BASIC gibt es keinen entsprechenden Befehl. Also bietet es sich an, ihn mit in das BASIC zu integrieren. Bei anderen BASIC-Dialekten wird er unter PRINT§XX,TT geführt, wobei XX für die Bildschirmnummer (beginnend bei 0) und TT für einen beliebigen auszudruckenden Text steht.
Gehen wir in gleicher Weise vor, dann heißt das in unserem Fall, daß wir vor jeder Abarbeitung des Maschinenmoduls den Wert XX aus dem BASIC-Text übernehmen müssen, den Cursor dann entsprechend setzen und ins BASIC an die Stelle zurückkehren, die uns den Ausdruck von TT erlaubt.

Wer sich mit dem Aufbau eines BASIC-Programms noch nicht so recht auskennt, der sollte sich dies erst einmal gründlich zu Gemüte führen. Hier wenigstens die wichtigsten Punkte, die zum Verständnis dieses Kapitels notwendig sind:

- In BASIC-Programmen sind Befehle und Zeichen codiert abgespeichert. Nach der Zeilennummer (LO/HI) folgen Codezahlen von 0 bis 255. Beispielsweise hat der Befehl PRINT die Nummer 153, PRINT# dagegen wird mit 152 verschlüsselt (deswegen kann man PRINT# nicht mit '?#' abkürzen). Diese Schlüsselzahlen für die einzelnen BASIC-Wörter bilden den sog. **Interpretercode.**

- Beim Ablauf eines Programmes wird nun Zeichen für Zeichen dieses BASIC-Textes von der RUN-Routine geholt und zunächst darauf untersucht, ob es sich bei der Code-Nummer um einen Befehl oder um ein sonstiges Zeichen (Zahl, Trennzeichen, String o.ä.)

handelt. Liegt ein Befehl vor, wird die Stelle im ROM gesucht, die diesen Befehl abarbeitet, und anschließend angesprungen.

- Wenn notwendig, werden aus dem BASIC-Text weitere Zeichen geholt (eben diese Parameter) und verarbeitet. Das macht die aktive BASIC-Routine nun selbst.
Denken wir nur daran, wenn der Befehl PRINT auftaucht. Dann wird ja meistens auch etwas ausgedruckt, nämlich das, was hinter dem Befehl PRINT bis zu einem Trenn- oder Endezeichen folgt.

- Damit der BASIC-Interpreter auch weiß, wo er gerade im Text ist, wird in der Zeropage ein Zeiger mitgeführt, der immer auf das eben behandelte BASIC-Zeichen des Programms zeigt.

- Die wichtigste BASIC-Routine ist nun die, die Zeichen für Zeichen aus dem BASIC-Programm holt und den eben besprochenen Programmzeiger verwaltet. Es ist die **CHRGET-Routine** (von get character), die als Unterprogramm von etlichen BASIC-Routinen benützt wird. Nämlich immer dann, wenn irgendwelche Parameter aus dem BASIC-Text in die gerade arbeitende Interpreter-Routine übernommen werden sollen.
Nun hat sich der Kreis geschlossen: Genau das wollen wir mit unseren Maschinenmodulen auch tun. Zapfen wir also wieder einmal die betriebsinternen ROM-Routinen an.

10.2 Zeichen aus dem BASIC-Text holen mit CHRGOT und CHRGET

Nehmen wir an, wir wollen das Zeichen 'x' aus dem BASIC-Programm entnehmen, das direkt hinter dem SYS-Aufruf steht.

Beispiel für eine BASIC-Zeile:

```
100 x=5000 : sys 20000 x : print x
```

Starten Sie diese Zeile nun mit RUN, dann wird nach dem Aufruf der SYS-Routine der Programmzeiger auf dem 'x' stehenbleiben. Das (A)-Register enthält jetzt den ASCII-Code für 'x'. Wir können ihn zur Überprüfung mit **CHROUT** ausgeben.

Um auch noch zu beweisen, daß der Programmzeiger tatsächlich auf 'x' steht, rufen wir die **CHRGOT**-Routine auf, die immer das ak-

tuelle Zeichen aus dem BASIC-Text holt, ohne den Programmzeiger weiterzustellen.

Das ASSEMBLER-Programm, das ab (19997) steht, muß nun folgendermaßen aussehen (Adressen für die 80XX-Serien in Klammern):

ASSEMBLER-Beispiel "87-basictext"

```
 19997 jsr 65490     ; CHROUT druckt Zeichen in (A)
 20000 jsr 121       ; CHRGOT-Routine holt das Zeichen, auf dem
(20000 jsr 118)        der Programmzeiger momentan steht, nach (A)
 20003 sta 1200      ; Das geholte Zeichen kann nun weiter verar-
(20003 sta 33500)    ; beitet werden, z.B. mit STA XXXX auf dem
                       Bildschirm ausgegeben werden.
```

Der Programmzeiger steht immer noch auf dem 'x'. Wendet man nun die **CHRGET**-Routine an, wird der Programmzeiger um eins erhöht, das nächste Zeichen, also der Doppelpunkt geholt, der mit Code 58 übersetzt wurde. Die Fortführung unseres kleinen Programms bebeweist das:

```
 20006 jsr 115       ; CHRGET holt Trennzeichen
(20006 jsr 112)
 20009 jsr 65490     ; Trennzeichen wird mit CHROUT ausgegeben
```

Nun steht der Zeiger nicht etwa vor dem 'P' von PRINT, sondern vor dem Code 153, dem Code für das gesamte Wort PRINT. Lassen wir uns das vorführen:

```
 20012 jsr 115       ; CHRGET holt Code von PRINT
(20012 jsr 112)
 20015 sta 2023      ; gibt re/un (A)-Inhalt aus (ein reverses y)
(20015 sta 34767)    ; der ASCII-Code ist nicht druckbar
```

Wenn wir jetzt mit RTS ins BASIC-Programm zurückspringen, wird nur noch x vorgefunden, was logischerweise zu einem SYNTAX ERROR führt.
Setzen wir jedoch den Programmzeiger wieder um einen Schritt zurück, nämlich vor das PRINT, dann kann dieser Befehl von BASIC wieder ausgeführt werden:

```
 20018 dec 122       ; LO des Programmzeigers um 1 zurück
(20018 dec 119)
 20020 lda 122       ; untersuchen, ob 0 unterschritten wurde
```

```
(20020 lda 119)
 20022 cmp #255      ; also 255 erreicht wurde
 20024 bne 20028     ; wenn nicht, nächste Zeile überspringen
 20026 dec 123       ; wenn ja, auch das HI-Byte des Programmzäh-
(20026 dec 120)        lers erniedrigen
 20028 rts           ; jetzt kann zu BASIC zurückgekehrt werden
```

Wenn Sie wissen wollen, wie Ihr BASIC-Programm codiert im RAM liegt, brauchen Sie nur eine Schleife aus CHRGETs und Bildschirmausgaben zu bilden, den Programmzeiger auf die gewünschte Stelle aufzusetzen und eine Abbruchbedingung einzubauen. Beispielsweise werden beim Ende eines BASIC-Programms drei 0-Bytes gesetzt.

Aufgabe:
Schreiben Sie ein Programm, das 1000 bzw. 2000 Zeichen (einen Bildschirm voll) eines BASIC-Programms im BASIC-Code zeigt.

Label	C64	40/80XX
CHRGOT	121=$79	118=$76
	momentanes BASIC-Zeichen ---> (A)	
CHRGET	115=$73	112=$70
	Zeichen aus BASIC-Text ---> (A)	
PRGPTR	122/123=$7a/7b	119/120=$77/78
	Programmzeiger LO/HI steht auf momentanem Zeichen	

10.3 Byte-Auswertung mit GETBYT und VALBYT

Um die Ziffernfolge einer Zahl in den entsprechenden Wert umzuwandeln, bedient man sich der Routinen **GETBYT** und **VALBYT**, sofern der Wert der zu übernehmenden Zahl zwischen 0 und 255 liegt.
Zur Klarstellung sei gesagt, daß eine im BASIC-Text als Ziffernfolge abgelegte Zahl hier in ein einziges Byte übernommen wird, während im BASIC-Text ja jede Ziffer ein eigenes Byte beansprucht.

Zur Verdeutlichung zwei Beispiele, die auch gleichzeitig etwas Wiederholung bieten:

Beispiel 1: Auswertung mit GETBYT und Ausgabe mit STA

Nehmen wir wieder eine BASIC-Zeile an, mit der wir unsere nachfolgende Maschinenroutine überprüfen können:

```
100  co=20000:sysco#48,49:pokel200,48:pokel201,49
```

Für einen reinen BASIC-Programmierer sieht das höchst seltsam aus; jedoch wissen wir bereits, daß man mit dem Programmzeiger einiges manipulieren kann.

Was soll nun in dieser Zeile passieren?
- Als erstes legen wir mit der Variablen co die Einsprungadresse bei 20000 für den folgenden SYS-Befehl fest.
- Nach dem Einsprung in unser Maschinenprogramm stehen die Parameter #48,49 bereit, wobei die Zeichen # und das Komma als Trennzeichen fungieren. Wir kommen darauf noch zurück.
- Die Zahl 48 (genauer gesagt die Ziffernfolge 4-8) und die Zahl 49 stellen im Bildschirmcode die Zeichen "0" und "1" dar und sollen vom Maschinenprogramm übernommen und ausgegeben werden.
- Anschließend erfolgt der Rücksprung ins BASIC und wir poken dort die gleichen Codezahlen zur Kontrolle auf den Bildschirm.

Das ASSEMBLER-Programm sieht dann so aus (80XX in Klammern):

Beispiel "88-getbyte"

```
- 20000 jsr 121     ; CHRGOT holt das Zeichen, auf dem der
 (20000 jsr 118)     Programmzeiger momentan steht
 (20003 sta 33500)
  20003 sta 1160    ; und gibt es auf dem Bildschirm aus.
```

Durch diesen letzten Befehl wissen wir, wo sich der Zeiger nach dem SYS 20000 befindet, nämlich ein Zeichen weiter auf dem Doppelkreuz #.

```
- 20006 jsr 47003   ; GETBYT holt nun alle folgenden Zeichen,
 (20006 jsr 51409)    bis eine Nichtziffer oder ein Trennzeichen
                      auftritt, in unserem Fall bis zum Komma.
```

Anschließend wird der Wert der Ziffernfolge bestimmt und im (X)-Register sowie in der 5.Stelle des FAC1 abgelegt. In unserem Fall enthält das (X)-Register also den Wert 48. Geben wir ihn auf dem Bildschirm zwei Stellen weiter aus:

```
  20009 stx 1162   ; druckt Zeichen für Code 48: "0"
 (20009 stx 33502)
- 20012 jsr 121    ; CHRGOT holt das Zeichen, auf dem der
 (20012 jsr 118)    Programmzeiger steht.
  20015 sta 1164   ; wir erkennen, daß er auf das Komma zeigt
 (20015 sta 33504)
- 20018 jsr 47003  ; GETBYT holt nun die nächste Ziffernfolge
 (20018 jsr 51409)  als Byte nach (X) und FAC1+4
  20021 stx 1166   ; druckt das entsprechende Zeichen: "1"
 (20021 stx 33506)
```

Wir wissen schon, daß der Zeiger auf dem Trennzeichen steht, so daß wir sofort ins BASIC-Programm zurückkehren können:

```
- 20024 rts
```

Wenn Sie das Programm "88-getbyte" eingetippt haben, kehren Sie am besten in den READY-Modus zurück und geben die BASIC-Zeile ein, die wir uns vorhin ausgedacht haben.

Nach dem Starten mit RUN erhalten Sie den Ausdruck: # 0 , 1
und darunter die gepoketen Zeichen: 0 1

Alles klar bis hierher?
Wenn ja, dann spielen Sie mal ein wenig sowohl mit der BASIC-Zeile, als auch mit dem ASSEMBLER-Programm.
Sie sollten hinterher folgendes erkennen:

- Als Trennzeichen können sich alle nichtnumerischen Zeichen verwenden, so lange Sie keinen Einfluß auf die Einsprungvariable CO haben:
  Damit verbieten sich die Zeichen "+","-","/","·" und die Vergleichsoperatoren "<","=" und ">".
  sysco+48,49... hätte dann einen Einsprung bei 20048 zur Folge und würde hinterher SYNTAX ERROR erzeugen.

- Auch mit den Klammern muß man sorgfältig umgehen:
  sysco(48) wird nämlich als Einsprung beim Wert der Feldvaria-

blen co(48) erkannt und führt bei uns garantiert zu einem ILLEGAL QUANTITY ERROR, weil wir co(x) nicht dimensioniert haben.

- Dagegen lassen sich die Klammern durchaus folgendermaßen verwenden: sysco)48(49... oder sys(co)(48),49...

- Die **GETBYT**-Routine stellt nach ihrem Abschluß den Zeiger immer auf das Trennzeichen, denn dort ist sie ja schließlich gelandet, um festzustellen, daß die Ziffernfolge für den Bytewert zu Ende ist.

- Werden Dezimalzahlen verwendet, dann arbeitet das Programm trotzdem normal. Es wird aber nur der Integerwert beachtet. Sie können statt 48 durchaus 48.0235 eingeben. Das Ergebnis ist das gleiche.

- Überschreiten Sie allerdings den Wert 255.999..., dann spielt diese ROM-Routine nicht mehr mit. (Wir lassen uns dazu aber im nächsten Abschnitt etwas einfallen.)
  Das Ergebnis ist dann wieder ein ILLEGAL QUANTITY ERROR.

Mit dem nun erworbenen Wissen läßt sich das nächste Beispiel wesentlich schneller besprechen.

Beispiel 2: Byte-Auswertung mit VALBYT und Ausgabe mit CHROUT

Die entsprechende BASIC-Zeile, anhand der wir unser Maschinenprogramm testen wollen, sieht fast genau so aus wie zuvor:

```
100 co=20000 : sys(co)147,35 : print"screen CLR und #"
```

Sie haben es erkannt: Die Einsprungadresse ist eingeklammert. Es gibt hier kein Trennzeichen zwischen dem SYS-Aufruf und dem ersten Parameter 147. Der Programmzeiger steht anschließend auf der Ziffer "1" von 147. Die Einsprungadresse (co) wird also von "Klammer auf" bis "Klammer zu" ausgewertet. (Auch darauf kommen wir noch zurück.)

Das ASSEMBLER-Programm "89-valbyte" (80XX in Klammern):

```
- 20000 jsr 47006    ; VALBYT wertet ebenfalls die kommende
 (20000 jsr 51412)    Ziffernfolge aus, beginnt aber mit dem ak-
                      tuellen Programmzeiger
```

Zur Erinnerung: **GETBYT** würde ein Zeichen weiter beginnen, also mit der Ziffer "4" von 147.

```
  20003 txa          ; der Bytewert wird nach (A) übertragen und
  20004 jsr 65490    ; mit CHROUT ausgegeben, also nicht als
 (20004 jsr 65490)     Bildschirmcode, sondern als ASCII-Code, was
                       in unserem Fall dem "Bildschirm clear" ent-
                       spricht

- 20007 jsr 121      ; CHRGOT holt das zuletzt geholte
 (20007 jsr 118)       Zeichen noch einmal nach (A)
  20010 sta 1162     ; Ausgabe: wie erwartet ","
 (20010 sta 33502)

- 20013 jsr 47003    ; jetzt muß GETBYT angewendet werden
 (20013 jsr 51409)
  20016 txa          ; Ergebnis nach (A)
  20017 jsr 65490    ; und Ausgabe mit CHROUT
                        Das ist das Zeichen '#'
- 20020 rts          ; Rücksprung zu unserer BASIC-Zeile
```

Nach dem Aufruf der BASIC-Zeile mit RUN wird also zunächst der Bildschirm gelöscht, dann ein Komma ausgegeben und anschließend ein '#'. Zum Schluß wird der Kommentar ausgedruckt.

Zusammenfassung:

- **GETBYT** erhöht zunächst den Programmzeiger und holt dann die Ziffernfolge des folgenden BASIC-Textes als Byte-Wert in das (X)-Register und die Adresse (FAC1 + 4).

- **VALBYT** beginnt mit dem aktuellen Programmzeiger und arbeitet ansonsten genauso wie **GETBYT**.

- Bei beiden Routinen steht nach dem Durchlauf der Programmzeiger auf dem Trennzeichen hinter der Ziffernfolge.

- Das (A)-Register enthält den ASCII-Code dieses Zeichens.

Label	C64	40/80XX
GETBYT	47003=$b79b Ziffernfolge ab PRGPTR+1 als Byte nach (X) und (FAC1+4)	51409=$c8d1
VALBYT	47006=$b79e Ziffernfolge ab PRGPTR als Wert nach (X) und (FAC1+4)	51412=$c8d4

10.4 Eine Anwendung: PRINT AT-Routine mit Fehlermeldung

In Kapitel 6 haben wir besprochen, wie man den Cursor beliebig auf dem Bildschirm positionieren kann. Wenden wir dies nun in Verbindung mit unseren Routinen GETBYT und VALBYT an und erstellen ein Maschinenprogramm, das uns - von BASIC aufgerufen - zu einer beliebigen Bildschirmadresse bringt.

Dabei legen wir folgendes Format fest:
Nach dem SYS-Aufruf folgt die Zeilennummer (0 bis 24) und anschließend, durch ein Komma getrennt, die Spaltennummer (von 0 bis 39 bzw. 79)
Die BASIC-Zeile, mit der wir unser Programm austesten können, sieht dann vielleicht so aus:

```
100 at=20000 : sys(at)13,30,"Testausdruck: ";:print"alles klar?"
```

Wir wollen also den Ausdruck in Zeile 13, Spalte 30 beginnen.

Aufbau und Ablauf der Routine "90-print at":

Aus der Form der Parameterschreibweise (sie entspricht dem Beispiel 2 aus 10.3) ergibt sich der Einsatz der GETBYT- und VALBYT-Routinen.
Jetzt müssen wir noch sicherstellen, daß die Zeilen- und Spaltenzahl im zulässigen Bereich bleibt. Das geschieht durch einen Vergleich mit den oberen Grenzwerten (negative Zahlen werden sowieso für Byte-Einträge nicht zugelassen). Wird einer der Definitionsbereiche verlassen, zapfen wir die betriebsinterne Fehlermeldung ILLEGAL QUANTITY ERROR an, die anschließend den READY-Modus wieder herstellt.

Vielleicht überrascht es Sie, daß vor dem zu druckenden String kein PRINT-Befehl steht?

Nachdem sowieso klar ist, daß etwas ausgedruckt werden soll, benützen wir einfach den Einsprung in die PRINT-Routine schon in unserem Maschinenmodul und sparen uns dadurch einen BASIC-Befehl. Dazu muß aber noch das erste Zeichen des zu druckenden Ausdrucks (in unserem Fall das erste Anführungszeichen) aus dem BASIC-Text geholt werden, was aber mit CHRGET kein Problem ist.

Das ASSEMBLER-Programm "90-print at" (C64):

```
01 20000 jsr 47006   ; VALBYT holt die erste Ziffernfolge
                       nach der Klammer und wertet Sie aus. Das
                       Ergebnis steht als Byte in (X).
02 20003 cpx #25     ; maximale Zeilenzahl
   20005 bcs 20027   ; überschritten ===> Fehlermeldung
   20007 stx 214     ; sonst Zeile nach CURZEI

03 20009 jsr 47003   ; GETBYT holt nächste Ziffernfolge nach
                       dem Komma und legt den Wert nach (X) ab.
04 20012 cpx #40     ; maximale Spaltenzahl
   20014 bcs 20027   ; überschritten ===> Fehlermeldung
   20016 stx 211     ; sonst Spalte nach CURSPA

05 20018 jsr 58732   ; CURPOS setzt den Cursor

06 20021 jsr 115     ; CHRGET holt nächstes Zeichen nach (A),
                       und stellt dabei den Programmzeiger auf
                       das Zeichen '"' ein.
   20024 jmp 43682   ; BPRINT druckt den Ausdruck im BASIC-
                       Text, auf dessen Anfang der Programmzeiger
                       steht.
                       Anschließend erfolgt Rücksprung ins BASIC.
07 20027 ldx #14     ; Fehlernummer 14
   20029 jsr 42042   ; wird mit ERROR ausgegeben (ILL. QUA..)
   20032 rts         ; Ende (READY-Modus)
```

Haben Sie "90-print at" (20000) eingetippt? Dann können Sie die oben angeführte BASIC-Zeile variieren und mit RUN starten. Wie immer, probieren Sie andere Beispiele aus:
Geben Sie ruhig mal sys(at)40,20 o.ä ein. Es muß dann die Fehlermeldung kommen.

Für die 40/80XX-Serien müssen Sie "90-print at" so umbauen, wie wir es in Kapitel 6 besprochen haben.

10.5 Ausgabe von Fehlermeldungen aus dem ROM

Die C64-Geräte kennen 29 solche Meldungen. Hier braucht man nur die Nummer der Meldung in das (X)- oder (A)-Register zu laden und die **ERROR**-Routine ab (42042) aufzurufen.

Im einzelnen sind dies:

1 too many files	16 out of memory
2 file open	17 undef'd statement
3 file not open	18 bad subscript
4 file not found	19 redim'd array
5 device not present	20 division by zero
6 not input file	21 illegal direct
7 not output file	22 type mismatch
8 missing filename	23 string too long
9 illegal device number	24 file data
10 next without for	25 formula too complex
11 syntax	26 can't continue
12 return without gosub	27 undef'd function
13 out of data	28 verify
14 illegal quantity	29 load
15 overflow	

Der JMP nach **ERROR=**(42042) gibt die entsprechende Meldung aus und schließt gleichzeitig alle eventuell geöffneten Kanäle. Wie in BASIC erfolgt dann auch der Programmabbruch.

Die 40/80XX-Geräte rufen ihre Fehlermeldungen mit Hilfe eines Index auf, der ins (X)-Register geladen wird. Hier gibt es zwei Tabellen. Die erste entspricht den oben genannten Fehlern von Nummer 10 bis 27. Die zweite bezieht sich auf weitere Peripheriemeldungen, wobei (X) die Belegung 0,9,14,23,32,36,45,50,61,65,77, 86,95,100,109,116,127,134,144,148,159,163,170,174,182,191,197,209 enthalten kann. Diese Arten werden mit JMP 62895 ausgegeben.

Label	C64	40/80XX
ERRXX	42082=$a462	46048=$b3e0
	Fehlermeldung ohne Schließen der Kanäle	
ERROR	42042=$a43a	46031=b3cf
	Fehlertyp in (X): Index(Offset) bzw. Fehlernummer schließt alle Kanäle, beinhaltet aber nicht CLOSE	
ERR80	---	62895=$f5af
	Fehlermeldungen Peripherie (nicht C64)	
BPRINT	43682=$aaa2	47805=$babd
	Einsprung in die PRINT-Routine, dazu Programmzeiger auf dem ersten Zeichen des zu druckenden Ausdrucks, alle folgenden werden selbständig geholt, ausgewertet und ausgegeben.	

10.6 Zur Schreibweise der BASIC-Befehle

Bisher haben wir uns um die Syntax (Schreibweise) der zu übernehmenden Parameter keine großen Gedanken gemacht. In unserem Modul ist es zunächst gleichgültig, welche Trennzeichen wir verwenden. Diese Großzügigkeit kann aber manchmal ihre Tücken haben, so daß man auch die Richtigkeit der vorgesehenen Trennzeichen überprüfen sollte.
Folgende ROM-Routinen stehen dafür zur Verfügung:

Label	C64	40/80XX
KLMAUF	44794=$aefa	48882=$bef2
	prüft, ob letztes Zeichen ein Klammer auf "(" war	
KLAMZU	44791=$aef7	48879=$beef
	prüft, ob letztes Zeichen ein Klammer zu ")" war	
KOMMA	44797=$aefd	48885=$bef5
	prüft, ob letztes Zeichen ein Komma "," war	
PRFZEI	44799=$aeff	48887=$bef7
	prüft, ob Zeichen in (A) als letztes im BASIC-Text stand	

Zu beachten ist bei allen diesen Routinen, daß anschließend immer ein CHRGET durchgeführt wird, d.h. das nächste Zeichen wird geholt und der Programmzeiger wird inkrementiert (um eins weitergeführt).

Zur Byte-Auswertung wird also anschließend an die Trennzeichen-Überprüfung die Routine VALBYT eingesetzt, die ja mit dem Zeichen anfängt, auf dem der Programmzeiger gerade steht.

Ein kleines Beispiel:

```
100 x=80:y=90:te=20000:syste#(x,y)"test"
```

ASSEMBLER-Programm "91-trennzeichen" (allg.):

Folgende Labels werden zusätzlich als Beispiele verwendet:
BSX = 1200 (80XX: 33500)
BSY = 1202 (80XX: 33502)
Das sind die Bildschirmstellen für die Kontrollausgabe.

```
- 20000 lda #35      ; Code für #
  20002 jsr PRFZEI   ; vorhanden? nein ===> SYNTAX ERROR
- 20005 jsr KLMAUF  ; "(" vorhanden?
  20008 jsr VALBYT  ; wertet Ziffernfolge nach dem
                      Trennzeichen "(" aus
  20011 stx BSX     ; Ausgabe des Werts als BS-Code zur Kontrolle
- 20014 jsr KOMMA   ; "," vorhanden?
  20017 jsr VALBYT  ; ja ---> Auswertung bis zum nächsten Trenn-
                      zeichen mit VALBYT
  20020 stx BSY     ; und Kontrollausgabe im Bildschirmcode
- 20023 jsr KLAMZU  ; ")" vorhanden?
  20026 jmp BPRINT  ; ja ===> mit BPRINT den folgenden String
                      auf dem Bildschirm ausgeben und zurück nach
                      BASIC springen
```

Wenn Sie die BASIC-Zeile starten, werden Sie feststellen, daß auch die Variablen x und y richtig ausgewertet werden. Voraussetzung ist aber immer noch, daß sie im Bereich von 0 bis 255 liegen.

10.7 Auswertung mit VAREAL (Wert Realzahl)

Entsprechend der Byte-Auswertung arbeitet die Routine **VAREAL.** Jetzt kann die Ziffernfolge jedoch eine beliebige Zahl sein oder durch einen mathematischen Term dargestellt werden, der Zahlen und/oder Variablen mit oder ohne Vorzeichen enthält. Wichtig ist lediglich, daß der Wert des Gesamtterms eine reelle Zahl darstellt.
Das Ergebnis wird im FAC1 abgelegt und kann von dort aus weiterverarbeitet werden.
Alles bisher Gesagte über Trennzeichen usw. gilt nach wie vor.

Beispiel dazu:

```
100 x=20:y=-0.02:vr=20000 :SYSvr,x/y-(100-5000): PRINT"DM"
```

Ab (20000) muß nun wieder die Auswertungsroutine stehen.
Sie soll z.B. den Wert bestimmen und auf dem Bildschirm ausgeben.

ASSEMBLER-Beispiel "92-realwert" (allg.):

```
- 20000 jsr KOMMA   ; "," vorhanden?
  20003 jsr 44426   ; VAREAL wertet den folgenden Ausdruck
 (20003 jsr 48516)    bis zum Trennzeichen ":" aus und legt ihn
                      im FAC1 ab
- 20006 jsr FLPOUT  ; druckt: 3900
  20009 rts         ; Rücksprung ins BASIC
```

Nach der SYS-Routine wird zur Kontrolle das "DM" gedruckt, um sicherzustellen, daß auch wieder mit dem BASIC-Programm fortgefahren wird.

Sie erkennen auch, daß die Klammern diesmal nicht als Trennzeichen fungieren, da sie ja Zeichen eines mathematischen Ausdrucks sind. Verwenden Sie also in diesen Fällen das Komma oder den Doppelpunkt, um eine saubere Trennung Ihrer Parameter zu erzielen.

10.8 Auswertung mit VALKLA

Eine Abkürzung des eben beschriebenen Verfahrens ist möglich, wenn man die Routine **VALKLA** (Wert eines Klammerausdrucks) verwendet. Sie überprüft, ob die Klammern gesetzt sind und wertet die Zeichenfolge innerhalb der Klammern aus, wobei hier auch weitere Klammerausdrücke entsprechend der mathematischen Hierarchie behandelt werden.

Beispiel:

```
100 kl=20000:SYSkl,(2000-(500/0.1)):print" Klammerausdruck"
```

Das ASSEMBLER-Programm "93-klammerwert" (allg./80XX in Klammern):

```
- 20000 jsr KOMMA    ; "," vorhanden?
  20003 jsr 44785    ; VALKLA wertet die Zeichenfolge bis zum
 (20003 jsr 48873)     Trennzeichen ":" aus und bringt den Wert
                       nach FAC1
- 20006 jsr FLPOUT   ; druckt: -3000
- 20009 rts
```

Noch einmal zur Wiederholung, weil Sie sich viel Ärger ersparen können: Das Komma nach der Variablen kl ist unbedingt notwendig, weil sonst eine Startadresse kl(x) ausgewertet wird. Wir wollen aber kein Feldelement angeben, sondern nur die Zahl 20000.

10.9 Auswertung von Integerzahlen mit VALINT und INTADR

Hätte man den Wert eines Ausdrucks gern als Integerzahl im HI/LO-Format, bedient man sich der ROM-Routine **VALINT**.

Das Ergebnis wird in die 4. und 5. Stelle des FAC1 abgelegt, also nach **FAC+3/FAC+4**, von wo es weiterbehandelt werden kann. Im folgenden Beispiel begnügen wir uns wieder mit der Bildschirmausgabe:

Beispiel:

```
100 it=20000:sysit,25000-20/0.007:print" Integer"
```

ASSEMBLER-Beipsiel "94-intwert" (allg./80XX in Klammern):

```
- 20000 jsr KOMMA    ; prüft auf ","
- 20003 jsr 45493    ; VALINT wertet den Ausdruck bis zum
 (20003 jsr 49888)     Trennzeichen ":" aus, wandelt ihn in das
                       Integerformat und legt das Ergebnis in
                       (97/98), also (FAC1+3/FAC1+4) ab.
- 20006 lda 100      ; FAC+3, also HI nach (A)
 (20006 lda 97)
  20008 ldx 101      ; FAC+4, also LO nach (X)
 (20008 ldx 98)
  20010 jsr INTOUT   ; druckt: 22142
  20013 rts
```

Voraussetzung, daß keine ERROR-Meldung auftritt, ist, daß der Gesamtwert den die Zeichenfolge vom Eintritt der Auswertung bis zum Trennzeichen darstellt, den Integerbereich (von 0 bis 32767,...) nicht überschreitet.

Treten Bruchzahlen auf, wird der Nachkommaanteil einfach ignoriert, eine Fehlermeldung findet nicht statt.
Ist das Gesamtergebnis negativ, erfolgt ein ILLEGAL QUANTITY ...

Eine zweite Möglichkeit bietet **INTADR**, das die Integerzahl - wie das Label andeuten soll - in das Adressformat, also LO/HI umwandelt.
Diese Routine stammt aus dem BASIC-System, wenn Programmzeilennummern aus dem Bildschirm übernommen werden.

Ein kurzes Beispiel für eine mögliche BASIC-Zeile:

```
500 sys20000,32768:print" = $8000"
```

Das folgende ASSEMBLER-Programm übernimmt die Zahl 32768 als Integerzahl und druckt sie aus.

ASSEMBLER-Beispiel "95-integadress" (C64):

```
- 20000 jsr KOMMA    ; "," vorhanden ?
  20003 jsr 43371    ; INTADR holt die Zeichenfolge nach dem
 (20003 jsr 47350)     Komma, wandelt sie in LO/HI um und legt sie
                       als Integerzahl (Adressformat) nach (20/21)
                       (40/80XX: nach 17/18)
```

```
- 20006 ldx 20      ; Übertragen von LO
 (20008 sta 251)    ; nach (251)
  20010 lda 21      ; und HI
 (20012 sta 252)    ; nach (252)
  20014 jmp INTOUT  ; zur Ausgabe mit INTOUT
```

Nach dem Start mit RUN 500 wird gedruckt: "32768 = $8000"

Verwendet man bei den 40/80XX-Geräten zur Ausgabe **ADROUT**=55063, dann erhält man die Hexzahl zur eingelesenen Dezimalzahl, wenn wenn LO in (251) und HI in (252) steht.

10.10 Auswertung mit VALPAR, VALSTR, PARFLG und TYPFLG

Die ROM-Routine **VALPAR** (Wert eines beliebigen Parameters) wird immer dann eingesetzt, wenn man zunächst nicht weiß, welcher Art der auszuwertende Parameter ist.
Er kann ein Zahlenwert sein und hier wieder Byte, Real- oder Integerzahl darstellen oder aber er tritt als String auf.
VALPAR untersucht zunächst einmal den auftretenden Parameter und notiert sich in zwei Zeropageadressen, was für ein Typ ihr vorgelegt wurde:

In der einen Adresse (nennen wir sie **PARFLG**) steht eine 0, wenn es sich um einen Zahlenausdruck handelt; bei einem String wird 255 gesetzt.

Wird ein Zahlenausdruck als Integerzahl erkannt, wird in der Adresse **TYPFLG** das 8.Bit gesetzt (gleich 128), andernfalls - nämlich bei einer reellen Zahl - steht hier eine 0.

Die Auswertung erfolgt entsprechend den schon besprochenen Spielregeln:
- Reelle Zahlenergebnisse werden im FAC1 abgelegt.
- Integerzahlen werden in reelle Zahlen umgewandelt und ebenfalls nach FAC1 gebracht. Daß es sich weiterhin um eine Integerzahl handelt, wird in **TYPFLG** registriert.
- Treten Strings auf, dann steht in (FAC1+3/FAC1+4) ein Zeiger, der auf den entsprechenden Stringdeskriptor zeigt.

Die schon besprochene Routine **VALKLA** enthält übrigens das Programm **VALPAR** und kann genau so eingesetzt werden.

Da wir schon etliche Zahlenparameter ausgewertet haben, versuchen wir uns im folgenden Beispiel an einem String.

Beispiel:

```
100 x$="pytretemarap":pa=20000:syspa,mid$(x$,3):print" "x$
```

ASSEMBLER-Programm "96-parwert" (C64):

```
- 20000 jsr KOMMA    ; prüft auf ","
  20003 jsr 44446    ; VALPAR wertet die Zeichenfolge nach dem
 (20003 jsr 48536)     Komma bis zum Trennzeichen aus
  20006 jsr 44431    ; STRTYP prüft PARFLG auf 255
 (20006 jsr 48521)
```

Nachdem die Routinen **VALPAR** und **STRTYP** festgestellt haben, daß ein String zum Auswerten vorliegt, suchen sie auch noch die Lage des Strings im RAM. Jedoch wird dabei nicht die Zeichenkette selbst gesucht, sondern die Anfangsadresse des Stringdescriptors. Sie wird im FAC abgelegt und zwar in der 4. und 5. Stelle, also in **FAC+3** und **FAC+4**.

Die Adressen (100/101) enthalten nun den Zeiger auf den Stringdeskriptor von mid$(x$,3). Dort steht als erstes die Länge des Strings, also 10. Holen wir uns dieses Byte ins (X)-Register:

```
- 20009 ldy #0
  20011 lda (100),y
  20013 tax
```

Nehmen wir - nur so zum Spaß - an, wir wollen den String von hinten nach vorn schreiben, dann sieht das so aus:

```
- 20014 iny
  20015 lda (100),y ; LO der Anfangsadresse des Strings holen
  20017 sta 94      ; und ablegen in FAC+0 (Beispiel)
  20019 iny
  20020 lda (100),y ; HI der Anfangsadresse des Strings holen
  20022 sta 95      ; und nach FAC+1 bringen
  20024 txa
  20025 tay         ; Länge des Strings als Zähler nach (Y)
```

```
- 20026 dey          ; Zähler um eins erniedrigen
  20027 lda (94),y   ; Zeichen aus dem String holen
  20029 jsr 65490    ; und ausgeben
  20032 tya          ; Zähler nach (A)
  20033 bne 20025    ; Ende erreicht? nein ===> weiter holen
  20035 rts
```

Es ist etwas umständlich, sich vorzustellen, daß nach dem Auswerten ein Zeiger auf einen Zeiger zeigt; aber mit etwas Übung klappt auch das.

Für eine Vereinfachung sorgt die ROM-Routine **VALSTR**. Sie legt nämlich die tatsächliche Anfangsadresse des Strings (nicht den Anfang des Stringpointers!) in (34/35) bzw. 80XX:(31/32) ab. Nennen wir diese Zeropageadressen **STRADR** (von Stringadresse).

Diese Anfangsadresse steht auch noch in (X/Y) mit LO/HI; die Länge des Strings läßt sich dem (A)-Register entnehmen.

Um die oben besprochene BASIC-Zeile 100 genauso ablaufen zu lassen wie vorher, genügt jetzt folgendes ASSEMBLER-Programm:

ASSEMBLER-Beispiel "97-stringwerte" (C64/ 80XX in Klammern):

```
- 20000 jsr KOMMA
  20003 jsr VALPAR
  20006 jsr 46755   ; VALSTR holt Stringanfang nach STRADR
 (20003 jsr 51125)    und Stringlänge nach (A)
- 20009 tay
  20012 dey  ...
```

Der Rest ist der gleiche wie im vorigen Programm (Ausdruck von hinten nach vorn).

10.11 Übernehmen einer BASIC-Variablen mit GETVAR

Häufig treten Fälle auf, wo man möglichst schnell - also in Maschinensprache - einer BASIC-Variablen einen Wert zuweisen oder sie anderweitig weiterverarbeiten muß.
Die ROM-Routine **GETVAR** entnimmt dem BASIC-Text die Variable, untersucht sie auf numerisch (hier wieder auf integer oder reell)

und liefert in (A/Y) mit LO/HI die Anfangsadresse der Variablen. Wurde die gesuchte Variable nicht gefunden (erstes Auftreten im Programmlauf), so wird sie im Variablenspeicherbereich eingerichtet.

Zum Abschluß dieses Kapitels lösen wir ein Problem, das sowohl diese Routine **GETVAR** benützt, als auch eine Wiederholung und Anwendung etlicher Abschnitte aus bisher Besprochenem enthält. Außerdem bietet das Maschinenprogramm (selbstverständlich wieder in Modulform) etwas, was in BASIC nicht möglich ist:
Es liest in eine Stringvariable alle Zeichen und Zahlen ein, ohne Rücksicht darauf, ob sie eventuelle Trennzeichen sind oder nicht.

Es handelt sich im folgenden also um eine Art INPUT-Befehl, aber um einen sehr komfortablen: Er liest nämlich von jedem angesprochenen Gerät ein, also vom Bildschirm, von Diskette, vom Band oder von der Tastatur. Und das in einem Supertempo, so daß Sie z.B. eine vollgeschriebene DIN A4-Seite in 5 oder 6 Sekunden vom Floppy übernehmen können. Versuchen Sie das mal in BASIC.

In BASIC-Spracherweiterungen sind manchmal solche Befehle enthalten. Man bezahlt dafür ganz ordentliche Preise. Wir machen uns so einen Befehl selbst. Nennen wir das Modul also "onstring" und gehen ans Werk.

10.12 Ein vielseitiges Modul: "98-onstring"

Aufgabe des Moduls und die notwendigen Parameter:

- Es muß ein Gerät angesprochen werden, das zuvor über eine logische Adresse LA geöffnet wurde. (Das kann durchaus in BASIC geschehen.)
  Verwenden wir dazu - wie in BASIC auch - ein Doppelkreuz # und dahinter die File-Nummer LA.

- Eine Stringvariable ist notwendig, in die alle Zeichen eingelesen werden können. Nennen wir sie einfach mal x$.
  Selbstverständlich kann das im Gebrauch dann auch eine Feldvariable sein.

- Um eine bestimmte Anzahl Zeichen zu übernehmen, geben wir auch die maximale Länge des Strings an mit ML.

- Schließlich bringen wir als Option noch ein Abbruchkriterium ein. Taucht der Code dieses Bytes auf, dann wird der String abgeschlossen. Das Abbruchzeichen selbst übernehmen wir nicht mehr. (Wem das nicht gefällt, der baut diese Stelle eben entsprechend um.)

- Falls dieses Modul später noch mit anderen Spracherweiterungen verknüpft werden sollte, kennzeichnen wir es mit ON.
  (Auf die Verknüpfungen kommen wir noch zurück.)

Nehmen wir an, der Einsprung in das Modul erfolgt bei Adresse MO, dann sieht der BASIC-Befehl in allgemeiner Form so aus:

sys(mo)(on#la,x$,ml,az)

Als Beispiel zum besseren Verständnis:
Nehmen wir an die Einsprungadresse für unseren Befehl sei 20001.
sys 20001 (on#3,pr$,50,46) bedeutet, daß von dem Gerät, das mit der logischen Adresse #3 geöffnet wurde, 50 Zeichen geholt und in die Variable PR$ eingelesen werden sollen. Tritt das Zeichen mit dem Code 46 (also ein Punkt) auf, wird vorher abgebrochen.
Wenn das Öffnen mit OPEN 3,3 erfolgte, dann werden ab der momentanen Cursorposition alle Zeichen eingelesen einschließlich Anführungszeichen, Doppelpunkt und Komma, die hinter dem Cursor stehen.

Der Aufbau (Ablauf) des Moduls "98-ONSTRING":

01 Auf Klammer prüfen und das nächste Zeichen auf den Code von ON (145) vergleichen.

02 Nächstes Zeichen holen und auf Code 35 von "#" vergleichen.

03 Die nächste Zeichenfolge als Byte auswerten (das ist die logische Adresse).
Anschließend wieder die Kommaprüfung durchführen.

04 BASIC-Variable aus dem Text lesen, die Anfangsadresse holen und diese Anfangsadresse in der Zeropage zwischenspeichern. Außerdem noch eine Prüfung durchführen, ob es sich bei der Va-

riablen auch wirklich um eine Stringvariable handelt. Sonst gibt's später Schrott!

(In BASIC 4 - nicht C64 - muß ein eventueller String mit dem gleichen Namen ungültig gemacht werden, damit die "Stringmüllbeseitigung" systemgemäß funktionieren kann.
Dazu wird am Ende eines nicht mehr benötigten Strings der sog. "Trailer" (das sind zwei zusätzliche Bytes) mit der Stringlänge und mit dem Wert 255 belegt.)

05 Erneute Kommaprüfung und Auswertung der nächsten Zeichenfolge als Byte. (Das ist jetzt die vorgegebene Stringlänge ML.) Diese Länge in einer freien Zeropageadresse (auch jede andere möglich) festhalten.

06 Das nächste Zeichen erst auf "Klammer zu" prüfen; denn falls man kein Abbruchzeichen wünscht, sollte man hier den Befehl abbrechen können.

07 Ansonsten wird die nächste Zeichenfolge als Byte ausgewertet. Auch die Form ASC(".") wird richtig interpretiert und mit 46 festgehalten. Das ist z.B. das Abbruchzeichen.
Den ASCII-Code in einer freien Adresse speichern.

08 Eingabe-Einheit mit Hilfe der Filenummer aktivieren.

09 Das Y-Register als Zähler mit 0 vorbelegen und ein Zeichen holen. Dieses Zeichen wird - wie alle folgenden - zunächst einmal in einem Kassettenpuffer zwischengespeichert.

10 Überprüfen, ob das Abbruchzeichen vorliegt.
Wenn ja, aus der Schleife hinausspringen.

11 Statusbyte überprüfen, ob eventuell das Ende der Datei oder ein Fehler eingetreten ist.

12 Wenn nein, überprüfen, ob die maximale Länge schon erreicht ist. Wenn nicht, weitere Zeichen holen.

13 Ansonsten maximale Länge speichern und den Eingabevorgang abschließen.

14 Nachdem alle Zeichen geholt wurden, wird der Zähler nun in jedem Fall nach (A) übertragen. Das ist nun die endgültige

Stringlänge. Mit dieser in (A) kann nun der notwendige Platzbedarf im Stringbereich eingestellt werden. Die entsprechende ROM-Routine heißt **STRPLZ** (Stringplatz). Dazu muß die gewünschte Stringlänge im Deskriptor der Variablen stehen.

Die Stringlänge wird noch einmal geholt und als Zähler für die Einleseschleife hergerichtet: Dazu muß dieser Zähler zunächst einmal erhöht werden, weil die folgende Schleife sofort mit einer Dekrementierung beginnt (Erniedrigung um 1).

15 Da wir uns vorhin die Anfangsadresse der Stringvariablen gemerkt haben, können wir nun die gefundene Stringanfangsadresse in den Deskriptor schreiben.
Damit ist die Variable endgültig eingerichtet.

16 Schleife zum Einlesen der zwischengespeicherten Zeichen aus dem Kassettenpuffer in den vorbereiteten Stringplatz.

(In BASIC 4.0 darf nicht vergessen werden, hinter die Zeichenfolge des Strings die Anfangsadresse des zugehörigen Variablennamens zu schreiben. Sonst kennt sich die "carbage collection"-Routine nicht mehr aus.)

Die Beschreibung des Ablaufplans ist diesmal absichtlich etwas feiner gegliedert, weil es sich lohnt, anhand dieses Beispiels den Einsatz der einzelnen ROM-Routinen und die Eigenarten der CBM-Stringverwaltung noch einmal einzustudieren.

Für den C64 müssen die eingeklammerten Teile entfallen, da dieser noch mit dem alten BASIC 2.0 arbeitet. In diesem Dialekt gibt es vom String selbst keinen Rückverweis auf die Anfangsadresse der Stringvariablen.

Verwendete Adressen der Zeropage:

<163> = Filenummer aus dem BASIC-Text
<165> = Code des Abbruchzeichens
<166> = maximale Länge der zu holenden Zeichenkette
<167/168> = Variablenname

Nun das vielseitige ASSEMBLER-Programm "98-onstring" (C64):

```
01 20000 rts          ; Rücksprung, falls nicht "ON" als Erken-
                        nungsbyte im Befehl auftritt
   20001 jsr 44794    ; KLMAUF prüft auf "("
   20004 cmp #145     ; nächstes Zeichen Code für "ON"?
   20006 bne 20000    ; zurück ins BASIC, SYNTAX ERROR folgt

02 20008 jsr 115      ; CHRGET holt nächstes Zeichen
   20011 lda #35      ; Code für "#"
   20013 jsr 44799    ; PRFZEI prüft, ob "#" vorhanden

03 20016 jsr 47006    ; VALBYT wertet nächste Zeichenfolge als
                        Byte aus
   20019 jsr 44797    ; KOMMA prüft auf ","
   20022 stx 163      ; (X) zunächst in (163) zwischenspeichern
         nop            (X) erhielt die Filenummer von VALBYT

04 20025 jsr 45195    ; GETVAR holt Variable aus dem Text und
                        richtet sie im Variablenbereich ein
   20028 jsr 44431    ; STRTYP prüft auf Stringvariable
   20031 sta 167      ; Variablenadresse LO zwischenspeichern
   20033 sty 168      ; und HI ebenfalls
         nop
05 20036 jsr 44797    ; Kommaprüfung und nächstes Zeichen holen
   20039 jsr 47006    ; VALBYT wertet nächste Zeichenfolge aus
   20042 stx 166      ; das ist die maximale Länge des Strings
                        ---> zwischenspeichern
06 20044 cmp #41      ; folgt Trennzeichen ")"?
   20046 beq 20056    ; ja ===> Sprung nach 08

07 20048 jsr 44797    ; KOMMA-Prüfung
   20051 jsr 47006    ; GETBYT holt nächste Zeichenfolge/Byte
   20054 stx 165      ; Abbruchzeichen zwischenspeichern
   20056 jsr 44791    ; KLAMZU: folgt Klammer zu?
         nop
08 20060 ldx 163      ; Filenummer holen
   20062 jsr 65478    ; und mit CHKIN Eingabe vorbereiten

09 20065 ldy #0       ; (Y) als Zähler
   20067 jsr 65508    ; GETIN holt Zeichen vom aktiven Kanal
                        nach (A)
```

```
10 20070 cmp 165      ; Abbruchzeichen?
   20072 beq 20088    ; ja ===> Sprung Ende Eingabe

11 20074 ldx 144      ; STATUS
   20046 cpx #64      ; auf Dateiende prüfen
   20078 beq 20088    ; erreicht ===> Sprung zu Eingabe Ende
   20080 sta 828,y    ; sonst Byte ablegen (Puffer)
   20083 iny          ; Zähler erhöhen
12 20084 cpy 166      ; maximale Länge erreicht?
   20086 bne 20067    ; nein ===> zur Einlese-Schleife

13 20088 sty 166      ; Länge des geholten Strings ablegen
   20090 jsr 65484    ; und mit CLRCH Eingabe abschließen
         nop
14 20097 lda 166      ; Länge nach (A)
   20099 ldy #0       ; Index für folgende Ausgabe = 0
   20101 sta (167),y  ; neue Länge des Strings in Descriptor
   20103 ldx 167      ; Adresse der Stringvariablen LO nach (X)
   20105 ldy 168      ; dto. HI für die Routine
   20107 jsr 46201    ; STRPLZ schafft Platz für den String

15 20110 ldy #1       ; Zähler 1, weil bei 0 sonst das folgende
                        Byte auf die Länge des Strings gesetzt
                        würde (im Descriptor)
   20112 lda 98       ; LO der Stringanfangsadresse
   20114 sta (167),y  ; in den Descriptor
   20116 iny          ; Zähler +1
   20117 lda 99       ; HI der Stringanfangsadresse
   20119 sta (167),y  ; in den Descriptor
         nop
16 20122 ldy #0       ; Zähler auf 0
   20124 lda 828,y    ; Byte aus dem Puffer holen
   20127 sta (98),y   ; und in den vorbereiteten Stringbereich
   20129 iny          ; Zähler +1
   20130 cpy 166      ; maximale Länge erreicht?
   20132 bne 20124    ; nein ===> weiter schreiben
   20134 rts          ; ja ---> fertig
```

Anmerkung:
Bei Strings ist sauber auseinanderzuhalten:

- der Variablenanfang, womit die Adresse LO/HI gemeint ist, die in den Variablenbereich des RAMs - also direkt hinter den BASIC-Text - zeigt. Dort steht erst der Stringdeskriptor. Der zweistellige Name der Variablen liegt direkt davor.

- der Stringanfang, womit die Adresse LO/HI gemeint ist, die auf den Anfang der Zeichenkette im oberen RAM-Bereich zeigt. Dieser Stringanfang steht als 2. und 3. Byte im Stringdeskriptor. Im ersten finden wir die tatsächliche Länge des Strings.
  (Siehe dazu auch nochmals Abschnitt 8.

10.13 "99-onstring80" für 40/80XX-Geräte

Um auch den 40/80XX-Besitzern nicht den Spaß zu verderben, haben wir uns die Mühe gemacht, auch für sie das "ONSTRING"-Modul zu erarbeiten, weil die Verwaltung der Strings sich deutlich vom "alten" BASIC 2.0 unterscheidet. Um ein funktionsfähiges Modul zu bieten, stellen wir es hier noch einmal komplett mit Kommentar für den 40/80XX vor:

ASSEMBLER-Programm "99-onstring80" (BASIC 4.0):

```
01 20000 rts          ; Rücksprung, falls nicht "ON" als Erken-
                        nungsbyte im Befehl auftritt
   20001 jsr 48882    ; KLMAUF prüft auf "("
   20004 cmp #145     ; nächstes Zeichen Code für "ON"?
   20006 bne 20000    ; zurück ins BASIC, SYNTAX ERROR folgt

02 20008 jsr 112      ; GETBYT holt nächstes Zeichen
   20011 lda #35      ; Code für "#"
   20013 jsr 48887    ; PRFZEI prüft, ob "#" vorhanden

03 20016 jsr 51412    ; VALBYT wertet nächste Zeichenfolge als
                        Byte aus
   20019 jsr 48885    ; KOMMA prüft auf ","

04 20022 jsr 65478    ; CHKIN macht IEEE-Bus empfangsbereit
                        (X) enthält die notwendige Filenummer
```

```
05 20025 jsr 49451   ; GETVAR holt Variable aus dem Text und
                       richtet sie im Variablenbereich ein
   20028 jsr 48521   ; prüft, ob Variable auch Stringvariable ist
   20031 sta 70      ; Variablenadresse LO
   20033 sty 71      ; und HI nach (70/71)

06 20035 ldy #0      ; (Y) als Index 0
   20037 lda (70),y  ; Stringlänge aus Deskriptor nach (A) holen
   20039 beq 20065   ; Länge 0 ===> Sprung zu Teil 07
   20041 pha         ; Stringlänge auf Stack retten
   20042 iny         ; <Y>=1 als Zähler
   20043 lda (70),y  ; LO des Stringanfangs holen
   20045 sta 31      ; nach (31) zwischenspeichern
   20047 iny         ; <Y>=2 als Zähler
   20048 lda (70),y  ; und HI des Stringanfangs holen
   20050 sta 32      ; und nach (32) bringen
   20052 pla         ; Stringlänge wieder vom Stack abheben
   20053 tay         ; Stringlänge nun als Index nach (Y)
   20054 sta (31),y  ; und als erstes Trailerbyte hinter die ur-
                       sprüngliche Zeichenkette setzen
   20056 iny         ; Zähler um 1 erhöhen
   20057 bne 20061   ; falls der Wert von 255 auf 0 springt
   20059 inc 32      ; HI-Byte der Stringanfangsadresse erhöhen
   20061 lda #255    ; 255 ist Zeichen für entwerteten String
   20063 sta (31),y  ; als 2. Trailerbyte setzen

07 20065 jsr 48885   ; Kommaprüfung und nächstes Zeichen holen
   20068 jsr 51412   ; VALBYT wertet nächste Zeichenfolge aus
   20071 stx 190     ; das ist die maximale Länge des Strings

08 20073 cmp #41     ; folgt Trennzeichen ")"?
   20075 beq 20082   ; ja ===> Sprung nach 09
   20077 jsr 51409   ; GETBYT holt nächstes Byte
   20080 stx 191     ; Abbruchzeichen nach (191)

09 20082 ldy #0      ; (Y) als Zähler
   20084 jsr 65508   ; GETIN holt Zeichen vom aktiven Kanal
                       nach (A)
   20087 sta 826,y   ; Zwischenspeichern im Kassettenpuffer

10 20090 cpy 190     ; maximale Länge erreicht?
   20092 beq 20107   ; ja ===> Sprung nach 15
```

```
l1 20094 cmp 191      ; Abbruchzeichen in (A)?
   20096 beq 20107    ; ja ===> Sprung nach 15

l2 20098 lda 150      ; Status prüfen (Ende?)
   20100 bne 20105    ; ungleich 0 ===> Sprung nach 14
   20102 iny          ; ansonsten Zähler erhöhen
   20103 bne 20084    ; und nächstes Zeichen holen, usw.
   20105 ldy #255     ; falls Status, also Dateiende ---> <Y>=255

l3 20107 tya          ; Zähler, also endgültige Länge des aufge-
                        nommenen Strings nach (A)
   20108 sta 190      ; und nach (190)
   20110 jsr 50590    ; STRPLZ stellt Stringplatz bereit, der
                        Anfang der neuen Zeichenkette befindet
                        sich anschließend in (95/96)
l4 20113 ldy 190      ; Stringlänge holen als Zähler
   20115 iny          ; und zunächst um 1 erhöhen
   20116 dey          ; Schleife zum Eintragen der Bytes beginnt
                        mit Erniedrigen des Zählers
   20117 lda 826,y    ; Zeichen aus Kassettenpuffer holen
   20120 sta (95),y   ; und an vorbereiteten Platz legen
   20122 tya          ; Übertragen nach (A) nur wegen Endeprüfung
   20123 bne 20116    ; Ende nicht erreicht ===> weitermachen
         nop
l5 20126 ldy #0       ; Index 0
   20128 lda 190      ; Länge holen
   20130 sta (70),y   ; und in den Stringdeskriptor eintragen
   20132 iny          ; Zähler erhöhen
   20133 lda 95       ; LO des Stringanfangs
   20135 sta (70),y   ; in den Stringdeskriptor
   20137 iny          ; Zähler erhöhen
   20138 lda 96       ; Hi des Stringanfangs
   20140 sta (70),y   ; in den Stringdeskriptor

l6 20142 ldy 190      ; Länge wieder holen
   20144 lda 70       ; Variablenanfang LO holen
   20146 sta (95),y   ; und als erstes Trailerbyte setzen
   20148 clc
   20149 inc 95       ; LO des Stringanfangs erhöhen
   20151 bcc 20155    ; kein Überlauf ===> eine Zeile überspringen
   20153 inc 96       ; ansonsten HI des Stringanfangs erhöhen
   20155 lda 71       ; Variablenanfang HI holen
   20157 sta (95),y   ; und als 2.Trailerbyte setzen
```

```
17 20159 jsr 48879    ; KLAMZU prüft auf ")", holt nächstes
                        Zeichen
   20162 jmp 48054    ; Abschluß mit CLALL - Rücksprung
```

Test des "ONSTRING"-Moduls mit "100-testonstring":

Schreiben wir nun ein kleines BASIC-Programm, das sowohl die Funktionstüchtigkeit des Moduls als auch seine Wirkungsweise bestätigt.
Belassen wir es zunächst im Bereich von (20000) bis (20165) und grenzen BASIC entsprechend mit HIMEM 20000 ab.
Zunächst überprüfen wir, ob es - wie versprochen - aus dem Bildschirm-RAM liest (Zeilen 100 bis 180).
Anschließend lesen wir ein paar Zeilen aus einer SEQ-Datei, die wir hier mit "seqtest" benannt haben. Sie haben sicher irgendeine solche Datei auf Diskette und können den entsprechenden Namen einsetzen (Zeile 200).
Sie beherrschen BASIC sicher so gut, daß sich ein zusätzlicher Kommentar erübrigt:

```
  50 poke 55,0:poke56,70 : poke51,0 : poke 52,70 :' himem/fretop
 100 dim x$(20):open3,3  : ' datei auf bildschirm
 110 ox=20001            : ' startadresse des moduls
 115 print chr$(19)"0123456789abcdefghijklm..."chr$(19);
 125 for n=0to19 : sys (ox)(on#3,x$(n),n+1)
 130 print chr$(19);     : ' home
 140 next n
 150 print: for n=0to19 : printx$(n) : next n : 'ausdrucke
 160 print (peek(51)+256*peek(52)) : ' stringzeiger
 170 for i=0to200 : next i : 'warten
 180 print chr$(147);:goto 115 : ' screen clr

 200 open8,8,8,"seqtest"
 210 for n=1to5
 220 sys20001(on#8,x$(n),n)
 230 next n
 240 close 8
 250 for n=1to5 : printx$(n): next n
 260 end
```

Starten Sie nun mit RUN, dann erscheinen auf dem Bildschirm immer größere Teilstücke der ersten Bildschirmzeile.

Mit dem Ausdruck des Stringzeigers (51/52) aus der Zeropage können Sie feststellen, daß auch die Stringmüllbeseitigung einwandfrei funktioniert: Der Zeiger nimmt einen immer kleiner werdenden Wert an, bis er an die Grenze des Variablenbereichs stößt. Nach der "carbage collection" springt er dann sofort wieder an seine Obergrenze.

Damit Sie diese Zeigerausgabe auch verfolgen können, haben wir eine kleine Warteschleife eingebaut (Zeile 155), bevor die nächste Ausgabenserie erfolgt.

Beginnen Sie mit RUN 200, dann läuft die Diskette an und holt die entsprechenden SEQ-Daten von der Diskette.

10.14 Verknüpfen von Modulen

Will man in einem Programm mehrere verschiedene Module aufrufen, so ist es umständlich, jedes einzelne mit SYSXXXX anzuspringen, weil jedes Modul eine andere Einsprungadresse hat.
Verkettet man jedoch die benötigten Maschinenprogramme, dann genügt eine SYS-Adresse. Voraussetzung ist, daß im Maschinenmodul selbst der richtige Sprung zum richtigen Befehl gefunden wird.

Nehmen wir an, wir haben die beiden Module "irq" und "him" zur Verfügung. Dann legen wir eine SYS-Adresse fest, bei der ein Einsprung von BASIC aus erfolgt, z.B. (20000) und legen eine Variable dafür an: mo=20000.

Wenn "irq" ein Modul ist, das den Interruptvektor auf eine bestimmte Adresse setzt, z.B. auf 18000, dann sieht der BASIC-Befehl etwa so aus:

```
sys(mo)(irq,18000)
```

Soll **HIM** ein Befehl sein, der die RAM-Obergrenze bestimmt, um eventuelle Maschinenprogramme zu schützen, können wir vielleicht so vorgehen:

```
sys(mo)(him,18000)
```

Die Unterscheidung der Befehle wird also nicht über die SYS-

Adresse getroffen, sondern über die Syntax dahinter.

Damit haben wir unser weiteres Vorgehen bereits festgelegt:
- Im ersten Teil des Maschinenprogramms muß abgefragt werden, ob die gewünschte Zeichenfolge des ersten Befehls existiert.

- Wenn ja, wird dieses Modul abgearbeitet.
  Wenn nicht, wird zum Beginn des nächsten Moduls gesprungen, wo wiederum zuerst auf richtige Schreibweise des Befehls kontrolliert wird.

- Man muß nicht unbedingt so streng wie das CBM-BASIC sein und bei kleinen Fehlern gleich das Programm zum Stillstand bringen. Vielleicht genügt es dem einen oder anderen bei "irq" nur auf "ir" zu prüfen oder gar bloß auf "i".
  Das ist jedoch Geschmacksache, worüber wir hier nicht streiten wollen.

- Folgen weitere Module, wird in derselben Manier weitergeprüft. Wird auch beim letzten Modul nicht die richtige Zeichenfolge gefunden, erfolgt der Rücksprung ins BASIC, was automatisch einen SYNTAX ERROR o.ä. auslöst; denn die nächsten Zeichen sind in aller Regel nicht BASIC-gerecht.

Schematisch läßt sich dieses Verfahren so darstellen:

```
Einsprung bei MO
Modul 1:              Zeichenfolge richtig?
            ja ---> Modul bearbeiten
          nein ===> Sprung zu Modul 2

Modul 2:              Zeichenfolge richtig?
            ja ---> Modul 1 bearbeiten
          nein ===> Sprung zu Modul 3
...

Modul N:              Zeichenfolge des letzten Moduls richtig?
            ja ---> letztes Modul bearbeiten
          nein ===> Rücksprung nach BASIC
```

Jetzt sollte man noch darauf achten, daß die Sprungbefehle nur aus BRANCH-Anweisungen bestehen. Es läßt sich dann ohne weiteres ein Modulpaket erzeugen, daß aus beliebig vielen - soviel der Speicherplatz eben zuläßt - Programmteilen besteht. Dieses Paket bringen wir dann beim entsprechenden BASIC-Programm unter, am besten im oberen RAM-Bereich.

Die Module können einzeln erstellt und abgespeichert werden. Verknüpft man mehrere miteinander, so schiebt man den ersten Befehl des nachfolgenden Moduls auf das RTS des vorhergehenden. Schauen wir uns dazu ein Beispiel an:

Modul 1: IRQ-Vektor setzen mit sys(mo)(irq,xx)

ASSEMBLER-Programm "101-irqset" (C64):

```
- 20000 ldx 122     ; Programmzeiger LO
  20002 stx 165     ; retten
  20004 ldx 123     ; Programmzeiger HI
  20006 stx 166     ; ebenso, falls nicht dieses Modul aufgerufen
                      werden soll
  20008 jsr 44794   ; KLMAUF "(" ?
  20011 cmp #73     ; nächstes Zeichen "i" ?
  20013 bne 20050   ; nein ===> Sprung zum Schlußteil
- 20015 jsr 115     ; nächstes Zeichen holen mit CHRGET
  20018 cmp #82     ; "r" ?
  20020 bne 20050   ; nein ===> Modulende
- 20022 jsr 115     ; nächstes Zeichen holen
  ...                 eventuell überprüfen (siehe oben)
  20025 jsr 115     ; nächstes Zeichen holen
  20028 jsr 44797   ; KOMMA "," ?

- 20031 jsr 43371   ; INTADR wertet Zeichenfolge als Integer-
                      zahl aus. LO/HI in (20/21)
  20034 sei         ; Interrupt abstellen
  20035 lda 20      ; LO des neuen IRQ-Vektors
  20037 sta 788     ; nach IRQ/LO
  20040 lda 21      ; HI ...
  20042 sta 789     ; nach IRQ/HI
  20045 cli         ; Interruptflag wieder zurücksetzen
  20046 jsr 44791   ; KLAMZU ")" ?
  20049 rts         ; und zurück nach BASIC
```

```
- 20050 lda 165      ; LO des Programmzeigers
  20052 sta 122      ; hinter SYS-Aufruf stellen
  20054 lda 166      ; HI des Programmzeigers
  20056 sta 123      ; ebenso
  20058 rts
```

Wenn wir das Verschieben von vornherein vermeiden wollen, dann beginnen wir das nächste Modul **HIM** am besten gleich mit der Adresse (20056). Jedes für sich kann aber einzeln abgespeichert werden.

Modul 2: RAM-Obergrenze festlegen mit sys(mo)(him,xx)

ASSEMBLER-Programm "102-himemset" (C64):

```
- 20058 lda 122 .... usw. wie oben (Retten des Programmzeigers)

- 20066 jsr KLMAUF  ; "(" ?
  20069 cmp #72     ; folgt "h" ?
  20071 bne 20106   ; Sprung ans Ende
    ... Zeichen holen und eventuelle weitere Überprüfungen ...
- 20092 jsr INTADR  ; holt Integerzahl nach LINNUM
  20095 lda 20      ; LO von LINNUM
  20097 sta 55      ; nach oberem RAM-Zeiger LO : MAXMEM-LO
  20099 lda 21      ; HI
  20101 sta 56      ; nach oberem RAM-Zeiger HI: MAXMEM-HI
  20103 jsr KLAMZU  ; ")" ?
  20106 rts         ; und zurück ins BASIC
```

Für weitere Verknüpfungen empfiehlt es sich, auch hier den Programmzeiger zwischenzuspeichern und eine entsprechende Ergänzung wie im obigen Beispiel anzuhängen.

Werden die Module länger, so kann es vorkommen, daß die Sprungweite den Wert 128 überschreitet. Um die verschiebbare Modulform beizubehalten, sind dann Zwischensprünge einzubauen, wie wir sie bereits kennengelernt haben.

Baut man sich einen ganzen Befehlssatz von größerem Umfang, so lohnt es sich manchmal, in die freien Steckplätze sog. SOFT-ROMs einzusetzen, die den BASIC-Arbeitsspeicher nicht belasten. (Beim C64 ist diese Möglichkeit ohnehin vorgesehen.)

Diese SOFTROMs haben gegenüber den EPROMs, wie sie z.B. für die Spracherweiterung EXBASIC im Handel sind, den entscheidenden Vorteil, daß sie jederzeit mit anderen Programmen (nur Maschinenprogrammen!) belegt werden können. Der Nachteil aber ist, daß sie von einer Speichereinheit aus erst geladen werden müssen, was jedoch mit einem Starterprogramm - wie wir gelernt haben - auch kein besonderes Problem ist.

10.14 Modulverknüpfung mit einer Sprungleiste

Wen es stört, daß bei jedem Einzelmodul zuerst der Programmzeiger gerettet und dann wieder zurückgesetzt werden muß, arbeitet am besten mit einer Sprungleiste für die verschiedenen Einsprungadressen.

Vorteil: Die Module selbst können stehen, wo sie wollen. Die Einsprungadressen müssen aber bekannt sein.
Nachteil: Eine Gesamtverschiebung ist nicht möglich.

Ein Beispiel dazu:
Nehmen wir an, unsere beiden Module "irqset" und "himemset" stehen ab (10100) bzw. (10200).
Der Einsprung ins Modulpaket soll bei (10000) erfolgen. Das ist also der SYS-Aufruf.

Jetzt wird im Sprungverteiler zuerst nach dem Buchstaben "i" gefragt. Ist er nicht vorhanden, wird gleich zur zweiten Modulüberprüfung verzweigt. Dort wird das gleiche erste Byte mit "h" verglichen usw.

Erst bei Gleichheit des ersten Buchstabens wird der zweite geprüft. Das spart Zeit.
Genauso fährt man fort, wenn mehrere Module hintereinander aufgerufen werden können: Erst wenn die ersten beiden Buchstaben passen, wird der dritte überprüft usw.
Für umfangreiche Modulpakete kann man sich dazu auch eine Prüfleiste ausdenken.

ASSEMBLER-Beispiel "103-modsprung" (allg.):

```
01 10000 jsr KLMAUF  ; "(" ?
   10003 cmp #73     ; "i" ?
   10007 bne 10030   ; nein ===> zur nächsten Modulprüfung
   10007 jsr CHRGET  ; nächstes Zeichen holen
   10010 cmp #82     ; "r" ?
   10012 bne 10037   ; nein ===> evt. 2. Zeichen im nächsten Teil
   10014 jsr CHRGET  ; nächstes Zeichen holen
   10017 cmp #81     ; "q" ?
   10019 bne 10044   ; nein ===> evt. 3. Zeichen im nächsten Teil
   10021 jsr CHRGET  ;  nächstes Zeichen nach (A)
   10024 jmp 10100   ; gefunden ---> Modul "irq" anspringen

02 10030 cmp #72     ; "h" ?
   10032 bne 10054   ; nein ===> Ende
   10034 jsr CHRGET  ; nächstes Zeichen
   10037 cmp #73     ; "i" ?
   10039 bne 10054   ; nein ===> Ende
   10041 jsr CHRGET  ; nächstes Zeichen
   10044 cmp #77     ; "m" ?
   10046 bne 10058   ; nein ===> Ende
   10048 jsr CHRGET  ; nächstes Zeichen nach (A)
   10051 jmp 10200   ; gefunden ---> Modul "him" anspringen
03 10054 rts         ;

04 10100 jsr INTADR .... Modul "irq"
   ...
05 10200 jsr INTADR .... Modul "him"
```

Auf diese Weise wird allerdings auch bei der Zeichenfolge "irm" oder "iim" ein Modul aufgerufen. Doch wenn dort nicht die richtigen Angaben stehen, erfolgt der SYNTAX ERROR.

Probieren Sie selbst aus, wie Sie Ihre Verknüpfungen anlegen können. Wir wollten Ihnen nur ein paar Anregungen geben.

Aufgaben:
Schreiben Sie je ein Modul für die Befehle DEEK und DOKE und verknüpfen Sie beide zu einer Einheit.

Wandeln Sie die bereits besprochene PRINTUSING-Routine entsprechend ab und verknüpfen Sie sie mit den bereits vorhandenen Befehlen (z.B. ON#, DEEK, DOKE, PRINT AT, IRQ, HIM und weiteren).

Anmerkung:
Der Befehl DEEK(X) erzeugt aus den Adressen (X/X+1) eine Integerzahl für <X>=LO und <X+1>=HI dieses Zahlenwerts.

Zusammenfassung:

Wenn Sie auf diese Art und Weise einen Befehlssatz erstellen, der Ihren Ansprüchen gerecht wird, können Sie Ihre folgenden BASIC-Programme recht schnell und bequem erstellen.

Einen Nachteil hat die ganze Sache noch: Ihre neuen, zusätzlichen Erweiterungen müssen alle mit SYS(MO) aufgerufen werden. Um neue BASIC-Worte direkt zu akzeptieren, bedarf es allerdings einiger Umbauten und Umleitungen. Das kostet beim Programmlauf aber Zeit, und gerade die wollten Sie doch mit Ihren Modulen einsparen.

Die SYS-Aufrufe der Module stellen eine brauchbare Lösung dar, eigene Befehle innerhalb eines BASIC-Programms einzusetzen. Verknüpft man die einzelnen Module, so kommt man mit einer einzigen Einsprungadresse aus.

Die Syntaxprüfungen können von den Maschinenprogrammen selbst übernommen werden. Dabei bleibt es dem Programmierer freigegestellt, wie scharf diese Prüfungen sind.

Beginnen z.B. zwei Module mit dem gleichen Buchstaben, so ist auf jeden Fall das erste und das zweite Zeichen zu untersuchen, damit es nicht zu Fehlinterpretationen kommt.

BASIC-Text-Routinen und -adressen

Label	C64	40/80XX
FAC1	97...102=$61...66	94...100=$5e...64
FAC+1/2	98/99=$62/63	95/96=$5f/60
FAC+3/4	100/101=$64/65	97/98=$61/62
STRADR	34/35=$22/23 Adresse des Stringanfangs nach VALSTR	31/32=$1f/20
VARADR	71/72=$47/48 Adresse des Variablenanfangs LO/HI nach GETVAR auch in (A/Y)	68/69=$44/45
LINNUM	20/21=$14/15 BASIC-Zeilennummer LO/HI	17/18=$11/12
PARFLG	13=$0d	07=$07
TYPFLG	14=$0e	08=$08

Label	C64	40/80XX
GETBYT	47003=$b79b BASIC-Text von (Programmzeiger +1) bis Trennzeichen als Byte-Wert ---> (X)	51409=$c8d1
VALBYT	47006=$b79e BASIC-Text von Programmzeiger bis Trennzeichen als Byte-Wert ---> (X)	51412=$c8d4
CHRGET	00115=$0073 erhöht Programmzeiger, BASIC-Textzeichen ---> (A)	00112=$0070
CHRGOT	00121=$0079 beläßt Programmzeiger, holt noch einmal das letzte BASIC-Textzeichen nach (A)	00118=$0076
VAREAL	44426=$ad8a BASIC-Text wird als reelle Zahl gemäß den Rechenregeln berechnet bis zum Trennzeichen ---> FAC1	48516=$bd84

VALINT 45493=$b1b5 49888=$c2e0
BASIC-Text wird bis Trennzeichen als Ganzzahl ausgewertet <FAC1+3/FAC1+4>=HI/LO

INTADR 43371=$a96b 47350=$b8f6
Ziffernfolge aus BASIC-Text als Integerzahl:
<20/21>=<$14/15> LO/HI <17/18>=<$11/12>

VALPAR 44446=$ad9e 48536=$bd98
wertet Zeichenfolge bis Trennzeichen als Zahl oder String aus:
Zahl ---> FAC1, <PARFLG>=0, <TYPFLG>=128(integer)
oder <TYPFLG>=0(reell)
String: Zeiger auf Deskriptor = <FAC1+3/FAC1+4>
<PARFLG>=255

VALSTR 46755=$b6a3 51152=$c7b5
Aufruf nach VALPAR
STRADR <34/35>=<$22/23> <31/32>=<$1f/20>
Stringlänge = <A>

VALKLA 44785=$aef1 48873=$bee9
wertet Zeichenfolge zwischen "(" und ")" aus wie VALPAR

GETVAR 45195=$b08b 49451=$c12b
liest Zeichenfolge als Variablenname, sucht sie oder legt sie an mit Variablenanfang LO/HI in
VARADR (71/72)=($47/48) (68/69)=($44/45)

GETFIP 57812=$eld4 62589=f47d
holt Dateinamen mit eventuell vorangestellter Laufwerknummer, Gerätenummer und eventuelle Sekundäradresse; setzt diese Fileparameter in die Zeropage
Standard-Interrupt-Einsprung

STRPLZ 46197=$b475 50590=$c59e
mit <A>=Stringlänge wird entsprechend neuer Platz im Stringbereich "angehängt".
<FAC1+1/FAC1+2>=Stringanfangsadresse

BPRINT 43682=$aab5 47805=$babd
druckt Zeichenfolge ab PRGPTR als String aus, dazu Programmzeiger auf erstes Zeichen im BASIC-Text

Trennzeichen - Syntax-Routinen

KLMAUF 44794=$aefa 48882=$bef2
<A>=40 ---> kein Syntaxfehler, Programmzeiger erhöhen und nächstes Zeichen ---> (A)
BASIC-Zeilennummer LO/HI

KLAMZU 44791=$aef7 48479=$beef
<A>=41 (Klammer zu)? - sonst wie oben

KOMMA 44797=$aefd 48885=$bef5
<A>=44 (Komma ?) - sonst wie oben

PRFZEI 44799=$aeff 48887=$bef7
<A> beliebig - sonst wie oben
BASIC-Text von (Programmzeiger +1) bis Trennzeichen

STRTYP 44431=$ad8f 48521=$bd89
<PARFLG>=255 ---> keine Fehlermeldung, also liegt ein String vor (nach der Auswertung mit VALPAR oder VALSTR o.ä.)

NUMTYP 44428=$ad8c 48519=$bd87
<PARFLG>=0 ---> keine Fehlermeldung, also liegt eine numerische Variable nach der Auswertung des BASIC-Textes vor

ERRXX 42082=$a462 46048=$b3e0
druckt Fehlermeldung

ERROR 42042=$a43a 46031=$b3cf
druckt Fehlermeldung, dazu
Offset Fehlernummer
in (X)
Alle Kanäle werden geschlossen, nicht jedoch die eventuell offenen Dateien.

In allen Fällen, in denen der BASIC-Text die geforderten und zur Überprüfung vorliegenden Zeichen nicht enthält, erfolgt eine Fehlermeldung entsprechend dem CBM-System.
Mit ERROR muß man dagegen die Art der Fehlermeldung durch die Belegung von (X) selbst wählen.

11

Diverse ROM-Hilfen – Anwendungen

11 Diverse ROM-Hilfen und Module

11.1 BASIC-Start vom Maschinenprogramm aus mit MRUN

BASIC-Programme oder -Programmteile lassen sich auch von einem Maschinenprogramm aus starten.
Dabei unterscheiden wir zwei Möglichkeiten:
Den Kaltstart, der alle Zeiger (auch die Variablenzeiger) zurücksetzt und den Warmstart, bei dem die Variablen erhalten bleiben.

Vor dem Start mit **MRUN** müssen die Zeiger entsprechend eingestellt werden. Auch das übernehmen die ROM-Routinen:

KALTPT stellt die Zeiger für einen Kaltstart,
WARMPT setzt sie für einen Warmstart.

Anschließend kann die Routine **MRUN** aufgerufen werden.

Beispiel für einen Warmstart "104-warmstart" (C64/80XX):

- jsr 42638 (jsr 46626) ; **WARMPT** richtet die Zeiger so ein, daß die Variablen erhalten bleiben.

- jsr 42948 (jsr 46943) ; **MRUN** startet das BASIC-Programm von der ersten Zeile an.

11.2 Warmstart mit MGOTO ab einer bestimmten Zeilennummer

Mit MGOTO läßt sich ein BASIC-Programm mit einer beliebigen Anfangszeile starten. Diese Zeilennummer muß aber existieren. Beispielsweise ist es möglich, diese Routine einzusetzen, um eine bestimmte Taste (oder Tastenfolge) zu einer Funktionstaste umzubauen, die es erlaubt, den Programmlauf abzubrechen und an einer vorher bestimmten Stelle, z.B. einem Menue, wieder aufzunehmen.

Beispiel:
Die Funktionstaste F2 soll so eingerichtet werden, daß sie das laufende BASIC-Programm abbricht und bei der BASIC-Zeile 2000 wieder aufnimmt.
Taste F2 hat die Nummer 4 und wird mit SHIFT bedient. Daher muß auch das SHIFT-Flag **SHIFLG** abgefragt werden.

Ablauf von "105-goto2000":

Setzen wir den IRQ-Vektor auf den Anfang unseres Moduls, dann müssen wir damit rechnen, daß der Abbruch mitten in einer Routine geschieht, die wiederum Unterprogramm der Interpreterschleife ist. Das kann zu Komplikationen führen.

01 Als erstes prüfen wir deshalb, ob das Ende einer BASIC-Zeile erreicht ist. Das letzte von der CHRGET-Routine geholte Zeichen muß eine 0 sein. Ist dies nicht der Fall, läuft der Standard-Interrupt ab (Teil 05).

02 Ist gerade ein BASIC-Zeilenende erreicht, wird geprüft, ob die Tasten SHIFT und F1 gedrückt sind.

03 Die anzuspringende Zeilennummer wird in **LINNUM** und vorsichtshalber auch in **BASLIN** bereitgestellt.

04 Jetzt kann die Routine **MGOTO** angelaufen werden.

05 Sprung zur Standard-Interrupt-Routine.

ASSEMBLER-Beispiel "105-goto2000" (C64/80XX):

```
01 20500 jsr 121      ; CHRGOT holt das Zeichen aus dem BASIC-
       (jsr 118)        text, auf dem der Programmzeiger gerade
                        steht
   20503 bne 20531    ; ungleich 0, also kein Zeilenende ===>
                        Sprung zum Standard-Interrupt (05)

02 20505 lda 203      ; KEY ? = welche Taste gedrückt?
       (lda 151)
   20507 cmp #4       ; Taste F1/F2 ? (nur C64)
   20509 bne 20531    ; nein ===> zu Teil 05
   20511 lda 654      ; ja ---> SHIFT-Flag SHIFLG prüfen
       (lda 152)
   20514 beq 20531    ; nicht gedrückt ===> Sprung zu Teil 5
```

```
03 20516 lda #208     ; LO-Byte von 2000
   20518 sta 20       ; nach LINNUM-LO
        (sta 17)
   20520 sta 57       ; nach BASLIN-LO
        (sta 54)
   20522 lda #7       ; HI von 2000
   20524 sta 21       ; nach LINNUM-HI
        (sta 18)
   20526 sta 58       ; und BASLIN-HI
        (sta 55)
04 20528 jsr 43171    ; MGOTO unterbricht das BASIC-Programm
        (jsr 47155)     und beginnt wieder bei Zeile 2000

05 20531 jmp 59953    ; IRQ - Routine (Standard)
        (jmp 58453)
```

Damit dieses Programm seinen Zweck erfüllt, müssen zwei Bedingungen gegeben sein:

- Der IRQ-Vektor steht auf 20100, wo "105-goto200" beginnt.
- Die BASIC-Zeile 2000 muß vorhanden sein.

Wenn Sie das "him"-Modul noch bei (20000) stehen haben, oder den Befehl "irq" zur Verfügung haben, kann ein Testprogramm etwa so aussehen:

Beispiel "106-gototest" - BASIC

```
   100 sys20000(irq,20100)
   200 print"test ";
   300 goto 200
   .... beliebige weitere Zeilen
  2000 print chr$(147)"goto 2000 erreicht"
  2100 print"test positiv"
```

Anmerkung:
Das Programm ist nicht hundertprozentig abgesichert gegen Fehlermeldungen.

Wollen Sie sich nicht auf eine feste Zeilennummer beschränken, dann programmieren Sie eben das Laden der Zeilennummer LO/HI über zwei freie Adressen, die Sie beliebig belegen können. Passen Sie aber auf, daß Sie nicht irgendwelche Zeropage-Adressen erwischen, die für den BASIC-Programmlauf von Bedeutung sind.

11.3 Startadresse einer BASIC-Zeile suchen mit BLINAD

Um die Adresse zu suchen, wo die Programmzeile mit einer bestimmten Zeilennummer beginnt, belegt man wie oben die Zeropage-Adressen **LINNUM** mit LO/HI der Zeilennummer.
Die Routine **BLINAD** (BASIC-Line-Adresse) sucht von Beginn des BASIC-Textes an nach dieser Nummer.
Wird sie gefunden, steht sie anschließend mit LO/HI in **LINAD**, ist sie nicht vorhanden, steht dort die Adresse der nächsthöheren Zeilennummer.

ASSEMBLER-Beispiel "107-basiczeile" (C64):

```
- lda #48      ; LO von Zeilennummer 8240
  sta 20       ; nach LINNUM-LO
  lda #32      ; HI von 8240
  sta 21       ; nach LINNUM-HI
- jsr 42515    ; BLINAD sucht den BASIC-Text nach dieser
  (jsr 46499)  ; Zeilennummer 8240 ab
                 Das Ergebnis steht beim C64 in (95/96) mit LO/HI
                 und beim 40/80XX in (92/93).
- lda 96       ; HI der gefundenen BASIC-Zeilennummer nach (A)
  ldx 95       ; LO nach (X) zur Ausgabe mit
  jmp 48589    ; INTOUT
 (jmp 53123)
```

11.4 Umschalten von Text- und Graphikmodus

11.4.1 Text/Graphik bei den 40/80XX-Geräten

Schauen wir uns ausnahmsweise zunächst die Geräte mit BASIC 4.0 an, weil es hier mehrere Möglichkeiten gibt:

Im Textmodus wird normalerweise zwischen zwei aufeinanderfolgenden Zeilen ein Zwischenraum ausgespart, während im Graphikmodus die nächste Zeile direkt an die vorhergehende angeschlossen wird.

Will man nun z.B. mit Graphikzeichen arbeiten, aber nicht auf die Abstände verzichten, dann setzt man den Textmodus mit **TEXMOD** und wählt anschließend den Graphikzeichensatz, indem man das Kontroll-Register **PCR** mit 12 belegt:

ASSEMBLER-Beispiel "108-texmod" (80XX):

```
- jsr 57368    ; TEXMOD schaltet auf Zwischenzeilen um
  lda #12      ; 12 laden
  sta 59468    ; und Peripherie Kontroll Register damit belegen
```

Auch der umgekehrte Fall ist möglich, nämlich Groß/Kleinschreibung ohne Zwischenräume:

ASSEMBLER-Beispiel "109-graphmod" (80XX):

```
- jsr 57371    ; GRAMOD schaltet 'ohne Zwischenräume'
- lda #14      ; 14 nach
  sta 59468    ; PCR
```

Probieren Sie auch aus, wie man wieder in den normalen Textmodus bzw. Graphikmodus zurückkommt!

11.4.2 Text/Graphik für C64

Bei den C64-ern entfallen die Routinen **TEXMOD** und **GRAMOD.** Es wird nur das Peripherie-Control-Register umgeschaltet:

```
- lda #23     ; "text", also Klein-/Großschrift
  sta 53272   ; nach PCR
  und
- lda #21     ; "graphik", also Normalmodus Großschrift/Graphik
  sta 53272
```

Das entspricht den BASIC-Befehlen poke 53272,21 usw.

11.5 Abfrage der STOP-Taste mit STOPRY und STOPO

Zum Aktivieren der STOP-Taste auch in Maschinenprogrammen müssen entsprechende Abfragen eingebaut werden.
Das kann auf mehrere Arten geschehen:

STOPRY kehrt bei gedrückter STOP-Taste in den READY-Modus zurück. Das Programm wird also abgebrochen.

STOPO setzt bei gedrückter STOP-Taste das Z-Flag (letzte Operation war gleich 0). Durch die BRANCH-Befehle BEQ oder BNE kann nun entsprechend den Bedürfnissen des Programms eine Verzweigung stattfinden.

11.6 Sprung in den READY-Modus mit MREADY

Das Aussteigen aus einem Maschinenprogramm geschieht durch Anlaufen der Routine **MREADY.**
Es erfolgt ein Abbruch des laufenden Programms mit Übergang in den READY-Modus und der entprechenden Bildschirmmeldung.

11.6 Verschieben von RAM-Bereichen mit TRABLO

Falls Ihr Assembler nicht bereits einen Transportbefehl zur Verschiebung eines beliebigen Datenblocks hat, läßt sich die ROM-Routine **TRABLO** einsetzen, die in BASIC Programmzeilen zwischen bereits vorhandene schiebt bzw. den Platz dafür schafft.

Dazu müssen folgende Adressen bereitgestellt werden:

BABL = Anfangsadresse des zu verschiebenden Blocks (Beginn des alten Blocks)
EABL1 = Endadresse +1 des alten Blocks
ENBL1 = Endadresse +1 des neuen Datenbereichs

Beispiel:
Das Programm "105-goto2000", das bei (20100) beginnt und bei

(20135) das letzte Befehlsbyte hat, soll so verschoben werden, daß es bei (17999) endet, also (18000)... nicht mehr berührt.

ASSEMBLER-Beispiel "llO-transblock" (C64):

```
01 15000 lda  #132     ; LO  von  20100
   15002 sta 95        ; nach BABL-LO
   15004 lda #78       ; HI von 20100
   15006 sta 96        ; nach BABL-HI

02 15008 lda  #168     ; LO  von  20136(!)
   15010 sta 90        ; nach EABL-LO
   15012 lda #78       ; HI von 20136(!)
   15014 sta 91        ; nach EABL-HI

03 15016 lda  #80      ;  LO von  18000(!)
   15018 sta 88        ; nach ENBL1
   15020 lda #70       ; HI von 18000 (!)
   15022 sta 89        ; nach ENBL1

04 15024 jmp 41919     ; TRABLO  verschiebt  den  Block in den
                         Bereich (17964)....(17999)
```

Selbstverständlich läßt sich auch ein Modul erstellen, dem man die Anfangsadresse des neuen Bereichs eingibt.
Dazu muß aus den Anfangs- und Endadressen des alten Bereichs die Blocklänge errechnet werden und zur Anfangsadresse des neuen Bereichs addiert werden.

Ein entsprechender BASIC-Befehl könnte die Form haben:

```
800 sys(mo)(tr,20100,20136,17964)
```

Aufgabe:
Erstellen Sie ein Maschinenprogramm in Modulform, das den oben genannten BASIC-Befehl entsprechend ausführt.

Bestimmt werden Sie diesen Befehl oft benützen, wenn Sie ihn erst einmal zur Verfügung haben. Die umständliche "Pokerei" entfällt dann endlich.
Wenn Ihr Assembler-Programm im RAM-Bereich liegt, haben Sie auch oft Schwierigkeiten, weil Sie vom Assembler aus keine Programme in den oberen RAM-Bereich verschieben können, ohne ihn zu vernichten. Auch hier hilft das Modul "transblock".

11.8 Abspeichern eines Datenbereichs mit MSAVE

Um einen Programmblock - sei er nun verschoben worden oder nicht - so abzuspeichern, daß er beim Laden wieder im selben Bereich eingelesen wird, müssen zunächst einmal Anfangs- und Endadresse des Blocks übergeben werden.

Erfolgt der Befehl von BASIC oder im Direktmodus, dann sind wie üblich der Dateiname und die Geräteadresse anzugeben.

Ein entsprechender Befehl könnte dann so ausschauen:

```
900 sysmo,15000,15030,"0:transblock",8
```

Der Ablauf des Moduls wäre dann folgender:

01 Erste Integerzahl holen und in **BEGDAT** (Programmanfangsadressen) abspeichern.

02 Zweite Integerzahl holen und in **ENDDAT** (Programmende-Adressen) ebenfalls LO/HI ablegen.

03 Aufruf der Routine **GETFIP**, die aus dem BASIC-Text die zur Abspeicherung notwendigen Parameter holt, nämlich den Programmnamen mit eventuell vorangestellter Laufwerknummer und die Geräteadresse.
Die entprechenden Zeropage-Adressen werden richtig belegt.

04 Aufruf der Routine **MSAVE**. Für den C64 gibt man dazu eine logische Adresse an (nicht notwendig bei 40/80XX).

Das ASSEMBLER-Programm "111-msave" (C64):

```
01 22000 jsr 44797 ; KOMMA "," ?
   22003 jsr 43371 ; INTADR holt Integerzahl nach LINNUM
   22006 lda 20    ; LO der Anfangsadresse
   22008 sta 193   ; nach BEGDAT-LO
   22010 lda 21    ; HI der Anfangsadresse
   22012 sta 194   ; nach BEGDAT-HI
```

```
02 22014 jsr 44797 ; KOMMA "," ?
   22017 jsr 43371 ; INTADR holt 2. Integerzahl
   22020 lda 20    ; LO
   22022 sta 174   ; nach ENDDAT-LO
   22024 lda 21    ; HI
   22026 sta 175   ; nach ENDDAT-HI

03 22028 jsr 44797 ; KOMMA "," ?
   22031 jsr 57812 ; GETFIP holt Datei-Parameter und belegt
                     damit die notwendigen Adressen

04 22034 lda #4    ; logische Adresse (Beispiel)
   22036 sta 184   ; nach LA
04 22038 jsr 62954 ; MSAVE speichert die Datei ab auf dem
                     angesprochenen Gerät
   22041 rts
```

Es empfiehlt sich, diesmal mit Kommata als Trennzeichen zu arbeiten, weil das der Syntax der benützten BASIC-Routinen entspricht. **GETFIP** führt nämlich selbst die Trennzeichenprüfungen durch.

Setzt man nun z.B. eine Klammer ")" als letztes Zeichen, dann wird ein SYNTAX ERROR erzeugt. Mit Hilfe des Programmzeigers könnte man das zwar vermeiden, jedoch wird dann die Routine umständlicher und länger. In solchen Fällen hält man sich an die Schreibweisen, wie man sie von BASIC her gewöhnt ist. Nur so ist ein rationeller Einsatz der ROM-Routinen erst möglich.

Aber auch hier gilt wie sonst auch: Wem das nicht gefällt, der entwerfe seine eigenen Strukturen.

11.9 Laden eines Programms mit veränderter Startadresse

Es kommt oft genug vor, daß man ein Maschinenprogramm mit dem Assembler erstellt und hinterher feststellen muß, daß man dieses Programm gern in dem Bereich hätte, wo der Assembler vorher war. Es wäre also praktisch, wenn wir ein Modul hätten, das uns ein Programm in einen nachträglich bestimmten Bereich lädt.

Normalerweise wird es von Diskette oder Band dorthin gebracht, wo die ersten beiden Bytes der Datei hinzeigen. Sie werden mit dem Programm abgespeichert.

Es bieten sich zwei Möglichkeiten an:

- Erstens kann man beim Abspeichern gleich den eben erwähnten Zeiger verändern, indem man vor dem "SAVE"-Vorgang das LO/HI der gewünschten Anfangsadresse über den IEEE-Bus ausgibt und anschließend verhindert, daß der momentane Programmanfang übertragen wird.
  Das hat aber den Nachteil, daß wir uns damit bereits auf eine zwar neue, aber festgelegte Adresse beschränken.
  Gerade das kann jedoch auch wieder ein Vorteil sein. Wir besprechen dieses Modul im nächsten Abschnitt 11.10.

- Die zweite Möglichkeit besteht darin, erst beim Ladevorgang die neue Anfangsadresse festzulegen.
  Damit sind wir variabel und können unser Programm oder Modul schieben, wohin wir wollen.

Diese zweite Art schauen wir uns etwas genauer an und erstellen ein Modul "112-posload", das folgende BASIC-Zeile ausführen kann:

```
1000 syslo,posload18000,"1:test",8
```

Im Klartext heißt das:
Das Maschinenprogramm, das mit der Adresse LO beginnt, soll das Programm "test" vom Diskettenlaufwerk 1 so laden, daß der erste Befehl bei Adresse (18000) beginnt (Anfangsposition 18000).

Struktur und Ablauf des Moduls "112-posload":

01 Kommaprüfung und Untersuchung der nächsten Zeichen auf den Code für POS und LOAD. Beides sind BASIC-Wörter und nehmen daher im BASIC-Text nur je ein einziges Byte ein.

02 Nächstes Zeichen nach (A) holen und die Ziffernfolge bis zum nächsten Komma als Integerzahl auswerten.

03 LO/HI dieser Integerzahl als Programmanfang in die entsprechenden Zeropageadressen ablegen.

04 Kommaprüfung und die Fileparameter des Programms holen und den richtigen Zeropageadressen zuordnen. (Keine Angst, wird alles durch eine einzige ROM-Routine, nämlich **GETFIP**, erledigt.)

05 LOAD-Flag setzen, **OPEN** und **TALK**, sowie Sekundäradresse ausgeben und eine logische Adresse bereitstellen.
- Die Routine **DIRPR** prüft auf den Modus und gibt im Falle des Direktmodus die Meldung "searching for ... " aus.

06 Den (IEC)-Bus aktivieren, das Floppy als TALKER aktivieren und den richtigen Kanal anwählen mit der Sekundäradresse 96.

07 Die ersten beiden Bytes aus der Datei holen - das sind LO und HI der ursprünglichen Anfangsadresse - und vernachlässigen.

08 In die entsprechende Stelle der LOAD-Routine einspringen - diesen Einsprung nennen wir **MLOAD** - und **TWAIT**-Routine durchlaufen.

09 Rücksprung zum aufrufenden Programm, also zurück ins BASIC hinter unsere eben eingelesenen Parameter.

Das ASSEMBLER-Programm zu "ll2-posload" (C64):

```
01 17000 jsr 44797   ; KOMMA "," ?
   17003 jsr 121     ; CHRGOT holt letztes Zeichen
   17006 cmp #185    ; war das Code für POS ?
   17008 bne 17077   ; nein ===> zurück nach BASIC
                       (erzeugt SYNTAX ERROR)
   17010 jsr 115     ; nächstes Zeichen mit CHRGET
   17013 cmp #147    ; Code für "LOAD"
   17015 bne 17077   ; nein ==ö=> Ende

02 17017 jsr 115     ; CHRGET holt nächstes Zeichen nach (A)
   17020 jsr 43371   ; INTADR wertet die nächsten Zeichen als
                       Integerzahl aus, legt sie nach (17/18)

03 17023 lda 20      ; LO
   17025 sta 174     ; nach PRGANF
   17027 lda 21      ; HI
   17029 sta 175     ; nach PRGANF-HI
```

```
04 17031 jsr 44797    ; KOMMA "," ?
   17034 jsr 57812    ; GETFIP holt nächste Zeichenfolge und
                        wertet sie als File-Parameter aus, belegt
                        Geräte-Adresse GA usw.

05 17037 lda #0       ; Loadflag
   17032 sta 147      ; setzen
   17041 lda #4       ; logische Adresse wählen
   17043 sta 184      ; und nach LA bringen
   17045 jsr 62895    ; DIRPR prüft auf Direktmodus
   17048 lda #96      ; Sekundäradresse für Laden (=0+96)
   17050 sta 185      ; nach SA

06 17052 jsr 62421    ; OPENI öffnet Bus mit den Filedaten
   17055 lda 186      ; Geräteadresse laden
   17057 jsr 60681    ; und Floppy (8) aktivieren mit TALK
   17059 lda 185      ; Sekundäradresse mit
   17061 jsr 60871    ; SASENT ausgeben

07 17065 jsr 60947    ; mit INBUS erstes Datei-Byte holen
                        (LO der alten Programmanfangsadresse)
   17068 jsr 60947    ; zweites Byte holen (=Anfangsadresse HI)

08 17071 jsr 62704    ; MLOAD lädt das Programm mit dem neuen
                        Zeiger aus PRGANF (174/175)
   17074 jsr 63213    ; TWAIT wartet Abschluß ab

09 17077 rts          ; Ende (oder Zeiger auf evt. Fehler)
                        Rücksprung ins BASIC
```

Der oben angeführte BASIC-Befehl wird sowohl im Direkt- als auch im Programmodus richtig durchgeführt.

Wie man leicht erkennen kann, lohnt es sich, die Maschinenprogramme zur BASIC-Unterstützung in Modulform zu schreiben, so daß ein beliebiges Verschieben - oder wie in unserem Fall ein Laden an eine andere Stelle - keine Probleme bereitet.

Übrigens wird mit dem eben vorgestellten Modul keiner der BASIC-Zeiger verstellt, so daß sofort nach Rückkehr aus der Laderoutine der BASIC-Text weiter abgearbeitet werden kann.

Die Befehlszeile nimmt natürlich auch Variablen. Das hat den Vorteil, daß die Einsprungadresse in das nachgeladene Maschinenpro-

gramm gleich mit berechnet werden kann, falls sie nicht identisch mit der Anfangsadresse ist.

Das Modul ist bereits mit einer Identität - nämlich mit "posload" - gekennzeichnet und kann daher leicht in ein Modulpaket eingebaut werden.
Wenn Sie sich erst einmal an diesen zusätzlichen Befehl gewöhnt haben, werden Sie ihn bei Ihrer Programmierarbeit nicht mehr missen wollen.

11.10 Datenblock mit variabler Anfangsadresse speichern

Wie in 11.9 angedeutet, kann man mit dem folgenden Modul ein Programm oder eine beliebige Datei auch so abspeichern, daß sie beim Laden in einem anderen Bereich erscheint als dem programmierten. Dazu muß der Ladezeiger **LDPTR** der SAVE-Routine in der Zeropage auf den gewünschten Wert eingestellt werden, bevor das Abspeichern selbst durchgeführt wird.

Ein Beispiel zur Verdeutlichung:
Nehmen wir an, das Programm "datei" steht von (15000) bis (20000) im RAM, soll aber später ab (27000) benützt werden.

Der Ladezeiger, das sind die ersten beiden Bytes einer Datei, darf demnach beim Abspeichern von "datei" nicht mit 15000 übernommen werden, sondern muß auf 27000 gestellt werden.
Bevor also die eigentliche Datei "datei" abgespeichert wird, müssen die Bytes 120/105 als LO/HI = 27000 gesendet werden.

Die BASIC-Befehlszeile soll dazu so aussehen:

```
1100  sa=18000:syssa,possave27000,15000,20000,"1:datei",8
```

Das heißt im Klartext:
Das Modul zum Abspeichern wird mit (18000) angesprungen und speichert alle RAM-Daten von (15000) bis (20000) unter dem Namen "datei" auf Diskettenlaufwerk 1 ab, wobei der Ladezeiger auf 27000 gesetzt wird.

Nach dem Laden von "datei" stehen die Daten von (27000) bis (32000) im RAM.

Nachdem wir bisher alle Module sehr ausführlich dokumentiert haben, genügt wohl das ASSEMBLER-Programm mit einer Kurzbeschreibung zum Verständnis.

ASSEMBLER-Programm für "113-possave" (80XX):

```
01 18000 jsr 44797   ; KOMMA "," ?
   18003 jsr 121     ; CHRGOT holt letztes Zeichen
   18006 cmp #185    ; Code für POS ?
   18008 bne 18104   ; nein ===> Ende
   18010 jsr 115     ; nächstes Zeichen nach (A) mit CHRGET
   18013 cmp #148    ; Code für SAVE ?
   18015 bne 18104   ; nein ===> Ende

02 18017 jsr 115     ; nächstes Zeichen mit CHRGET holen
   18020 jsr 43371   ; GETADR holt Integerzahl
   18023 lda 20
   18025 sta 172     ; LO als LDPTR-LO speichern
   18027 lda 21
   18029 sta 173     ; HI als LDPTR-HI speichern

03 18031 jsr 44797   ; KOMMA "," ?
   18034 jsr 43371   ; nächste Integerzahl nach LINNUM holen
   18037 lda 20
   18039 sta 193     ; LO nach PRGANF-LO
   18041 lda 21
   18043 sta 194     ; HI nach PRGANF-HI

04 18045 jsr 44797   ; KOMMA "," ?
   18048 jsr 43371   ; letzte Integerzahl holen
   18051 lda 20
   18053 sta 174     ; LO nach PRGEND-LO
   18055 lda 21
   18057 sta 175     ; HI nach PRGEND-HI

05 18059 jsr 44797   ; KOMMA "," ?
   18062 jsr 57812   ; GETFIP holt Fileparameter, setzt sie

06 18065 lda #97     ; Sekundäradresse zum Abspeichern
   18067 sta 185     ; nach SA
   18069 lda #4      ; logische Adresse (Beispiel)
   18071 sta 184     ; nach LA bereitstellen
   18073 jsr 62421   ; OPENI öffnet (IEC)-Bus
   18076 lda 186     ; Geräteadresse aus GA laden
```

```
   18078 jsr 60684    ; LISTN aktiviert Gerät als Empfänger
   18081 lda 185      ; Sekundäradresse ausgeben
   18083 jsr 60857    ; mit SASENL

07 18080 lda 172      ; Ladezeiger LDPTR-LO holen
   18088 jsr 60893    ; OUTBUS gibt ihn als erstes Byte aus
   18091 lda 173      ; Ladezeiger LDPTR-HI holen
   18093 jsr 60893    ; und als zweites Byte ausgeben

08 18096 jsr 64398    ; TRPSET setzt Transportzeiger für den
                        Abspeichervorgang mit Hilfe von LDPTR
   18099 ldy #0       ; Index 0 setzen für DATOUT
   18101 jsr 63017    ; DATOUT gibt alle Daten auf den Bus aus
                        von PRGANF=(193/194) bis PRGEND=(174/175)
   18104 rts          ; Rücksprung ins BASIC
```

Der Abschluß der Übertragung, also Schließen der Datei usw. wird automatisch mit von der Routine **DATOUT** erledigt. Schließlich ist das nur ein Einsprung in die CMB-Routine "SAVE" gewesen.

Falls Sie sich gewundert haben, daß wir zur Erkennung des Moduls nur zwei Zeichen geholt und untersucht haben: SAVE wird als Byte mit dem Code 148, POS wird als Byte mit dem Code 185 im BASIC-Text abgelegt. Beides sind ja BASIC-Wörter.

Eine Zusammenstellung der Einsprungadressen zu diesem Kapitel finden Sie - auch für die 40/80XX-Geräte - am Ende der ROM-Routinen-Liste von Kapitel 13.

Wir haben Ihnen nun die Möglichkeit gegeben, sich in die Maschinenprogrammierung einzuarbeiten. Sie sind nun in der Lage, alle Ihre Programme in ASSEMBLER zu schreiben und mit Hilfe der ROM-Routinen sehr schnell laufen zu lassen.

Sicher werden Sie immer wieder in den Listen auf den folgenden Seiten blättern müssen, aber Sie werden sicher rasch Fortschritte sehen.

Viel Spaß!

12

ASSEMBLER-Kurzschule

12 ASSEMBLER-Kurzschule

12.1 Die Register des Mikroprozessors 65XX

Register	Abkz.	Bits
Akkumulator	(A)	8
X-Register	(X)	8
Y-Register	(Y)	8
Programmzähler	(PC)	16
Stackpointer	(S)	8
Statusregister	(P)	8

12.2 Das Prozessor Statusregister P

Die acht Bits (0 bis 7) haben Flag-Funktion.

Bit	Flagbezeichnung	Abkz.	gesetzt bei
0	Carry-Flag	C	Übertrag
1	Zero-Flag	Z	Ergebnis = 0
2	Interrupt disable-Fl.	I	Interrupt gesperrt
3	Dezimalflag	D	Dezimalmodus
4	Break-Flag	B	nach Break-Befehl
5	---	-	---
6	Overflow-Flag	V	Bit 7=1 im Ergebnis
7	Negativ-Flag	N	Ergebnis negativ (Bit 7)

12.3 Beeinflussung der Flags durch Befehle

C-Flag:
ADC, ASL, CLC, CMP, CPX, CPY, LSR, PLP, ROL, ROR, RTI, SBC, SEC

Z-Flag und N-Flag:
ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, RTI, SBC, TAX, TAY, TSX, TXA, TYA

D-FLAG:
SED, CLD

B-Flag:
BRK

V-Flag:
ADC, BIT, PLP, RTS, SBC, CLV

I-Flag:
CLI, SEI

Die unterstrichenen Befehle beeinflussen direkt das entsprechende Flag, d.h sie setzen bzw. löschen es
Flag gesetzt: 1 Flag gelöscht: 0

12.4 Der Befehlssatz in ASSEMBLER

Alle ASSEMBLER-Befehle werden mit allen möglichen Adressierungsarten dargestellt.
Die Flagveränderungen sind bei den Beispielen nicht vollständig. Sie werden nur bei den wichtigsten Operationen mit angegeben. Ansonsten gilt die Aufstellung in Abschnitt 12.3.
Wir vereinbaren zur Beschreibung der ASSEMBLER-Befehle folgende Abkürzungen:

```
 #B   = Byte (unmittelbar)
  M   = Adresse (absolut)
  Z   = Zeropage-Adresse
 <M>  = Inhalt der Adresse M
 <Z>  = Inhalt der Zeropage-Adresse Z
 M,x  = Adresse M+x
 M,y  = Adresse M+y
<M,x> = Inhalt der Adresse (M+x)
<M,y> = Inhalt der Adresse (M+y)
(Z,x) = Adresse (Z+x/Z+1+x) ; Zweibyte-Adresse LO/HI
(Z),y = Adresse (Z/Z+1)+y   ; Zweibyte-Adresse LO/HI
<Z,x> = Inhalt von Adresse (Z+x/Z+x+1)
<Z>,y = Inhalt von Adresse (Z/Z+1)+y
 <A>  = Inhalt des (A)-Registers
 <X>  = Inhalt des (X)-Registers
```

```
 <Y>  = Inhalt des (Y)-Registers
 <c>  = Inhalt des C-Flags
 <n>  = Inhalt des N-Flags
 <z>  = Inhalt des Z-Flags
```

Für die Beispiele verwenden wir folgende Belegungen:

```
      #B   = 2
      Z    = 20
      M    = 5130
     <P>   = 00000001 = 1
  Stapel   = 199 / ... / ...
    <20>   = 10
    <21>   = 20
    <50>   = 100
    <60>   = 14
    <61>   = 16
    <70>   = 120
  <5130>   = 255
  <5170>   = 200
  <5180>   = 210
  <4110>   = 220
     <A>   = 5
     <X>   = 40
     <Y>   = 50
```

12.4.1 Eingabebefehle (Ladebefehle)

Bei den einzelnen Befehlen werden alle Adressierungsarten aufgeführt, die möglich sind.
Bei den Beispielen sind die Schreibweisen in der ASSEMBLER-Sprache nicht möglich, wenn ein Fragezeichen vorangestellt ist.

L D A : Lade (A) mit Inhalt der angesprochenen Adresse

```
Flags: Z,N
lda #B          ; lda #2 ---> <A>=2
lda Z           ; lda 20 ---> <A>=10
lda M           ; lda 5130 ---> <A>=255
lda Z,x         ; lda 20,x = lda 60 ---> <A>=14
lda M,x         ; lda 5130,x = lda 5170 ---> <A>=200
```

```
lda M,y        ; lda 5130,y = lda 5180 ---> <A>=210
lda (Z,x)      ; lda (20,x) = ?lda <60/61> = lda 4110 ---> <A>=220
lda (Z),y      ; lda (20),y = lda 5130,y = lda 5180 ---> <A>=210
```

L D X : Lade (X) mit Inhalt der angesprochenen Adresse

```
Flags: Z,N
ldx #B         ; ---> <X>=2
ldx Z          ; ---> <X>=10
ldx M          ; ---> <X>=255
ldx Z,y        ; ldx 20,y = ldx 70 ---> <X>=120
ldx M,y        ; ldx 5130,y = ldx 5180 ---> <X>=210
```

L D Y : Lade (Y) mit Inhalt der angesprochenen Adresse

```
Flags: Z,N
ldy #B         ; ---> <Y>=2
ldy Z          ; ---> <Y>=10
ldy M          ; ---> <Y>=255
ldy Z,x        ; ldy 20,x = ldy 60 ---> <Y>=14
ldy M,x        ; ldy 5130,x = ldy 5170 ---> <Y>=200
```

P L A : Stapelbyte nach <A>

```
Flags: N,Z
pla        ; ---> <A>=199
```

P L P : Stapelbyte nach <P>=Statusregister

```
Flags: wie geholtes Byte
plp        ; ---> <P>=199
```

12.4.2 Ausgabebefehle (Speicherbefehle)

S T A : <A> in angesprochener Adresse abspeichern

```
sta Z      ; sta 20 ---> <20>=5
sta M      ; sta 5130 ---> <5130>=5
sta Z,x    ; sta 20,x = sta 60 ---> <60>=5
sta M,x    ; sta 5130,x = sta 5170 ---> <5170>=5
```

```
sta M,y    ; sta 5130,y = sta 5180 ---> <5180>=5
sta (Z,x)  ; sta (20,x) = ?sta (60/61) = sta 4110 --->&lt;4110&gt;=5
sta (Z),y  ; sta (20),y = ?sta (5130+50) = sta 5180 --->
                 <5180>=5
```

S T X : <X> in angesprochener Adresse abspeichern

```
stx Z      ; ---> <20>=40
stx M      ; ---> <5130>=40
stx Z,y    ; stx 20,y = stx 70 ---> <70>=40
```

S T Y : <Y> in angesprochener Adresse abspeichern

```
sty Z      ; ---> <20>=50
sty M      ; ---> <5130>=50
sty Z,x    ; sty 20,x = sty 60 ---> <60>=50
```

P H A : <A> auf Stapel

```
pha        ; ---> Stapel = 5/199/.../...
```

P H P : <P> auf Stapel

```
php        ; ---> Stapel = 1/199/.../...
```

12.4.3 Arithmetische Verknüpfungen

A D C - Addition mit <A>

Das Carryflag muß vor dem ADC-Befehl gelöscht werden mit CLC.

```
Flags: C,Z,N
adc #B     ; adc #2 ---> <A>=7
adc Z      ; adc 20 = adc #10 ---> <A>=15
adc M      ; adc 5130 = adc #255 ---> <A>=4                <C>=1
adc Z,x    ; adc 20,x = adc 60 = adc #14 ---> <A>=19
adc M,x    ; adc 5130,x = adc 5170 = adc #200 ---> <A>=205
adc M,y    ; adc 5130,y = adc 5180 = adc #210 ---> <A>=215
adc (Z,x)  ; adc (20,x) = ?adc <60/61> = adc 4110 ---> <A>=225
adc (Z),y  ; adc (20),y = adc 5130,y = adc 5180 ---> <A>=215
```

S B C - Subtraktion von <A> mit Carryflag
Das Carryflag muß vor dem SBC-Befehl gesetzt werden mit SEC.

```
Flags: C,Z,N
sbc #B     ; sbc #2 ---> <A>=3
sbc Z      ; sbc 20 = sbc #10 ---> <A>=251                    <C>=0
sbc M      ; sbc 5130 = sbc#255 ---> <A>=6                    <C>=0
sbc Z,x    ; sbc 20,x = sbc 60 = sbc #14 ---> <A>=247         <C>=0
sbc M,x    ; sbc 5130,x = sbc 5170 = sbc #200 ---> <A>=61     <C>=0
sbc M,y    ; sbc 5130,y = sbc 5180 = sbc #210 ---> <A>=51     <C>=0
sbc (Z,x)  ; sbc (20,x) = sbc 4110 = sbc #220 ---> <A>=41     <C>=0
sbc (Z),y  ; sbc (20),y = sbc 5130,y = sbc 5180 ---> <A>=51   <C>=0
```

I N C - Erhöhung eines Speicherinhalts um 1

```
Flags: Z,N
inc Z      ; inc 20 ---> <20>=11
inc Z,x    ; inc 20,x ---> inc 60 ---> <60>=15
inc M      ; inc 5130 ---> <5130>=0                           <Z>=1
inc M,x    ; inc 5130,x = inc 5170 ---> <5170>=201
```

I N X - Erhöhung des X-Registers um 1

```
Flags:Z,N
inx        ; ---> <X>=41
```

I N Y - Erhöhung des Y-Registers um 1

```
Flags: Z,N
iny        ; ---> <Y>=51
```

D E C - Erniedrigen eines Speicherinhalts um 1

```
Flags: Z,N
dec Z     ; dec 20 ---> <20>=9
dec Z,x    ; dec 20,x = dec 60 ---> <60>=13
dec M      ; dec 5130 ---> <5130>=254
dec M,x    ; dec 5130,x = dec 5170 ---> <5170>=199
```

D E X - Erniedrigen des X-Registers um 1

```
Flags: Z,N
dex       ; ---> <X>=39
```

D E Y - Erniedrigen des Y-Registers um 1

```
Flags: Z,N
dey       ; ---> <Y>=49
```

12.4.4 Logische (bitweise) Verknüpfungen

Die folgenden Operationen finden im (A)-Register statt.
Ausnahme: BIT beeinflußt (A) nicht, sondern setzt nur die entsprechenden Flags.

A N D : 1 and 1 = 1; 1 and 0 = 0; 0 and 1 = 0; 0 and 0 = 0
geeignet zum Löschen einzelner Bits

```
Flags: Z,N
and #B    ; and #2 :     00000101 and 00000010 = 00000000
and Z     ; and 20 :     00000101 and 00001010 = 00000000
and Z,x   ; = and 60 = and #14 : 00000101 and 00001110 = 00000100
and M     ; = and #255 : 00000101 and 11111111 = 00000101
and M,x   ; = and 5170 : 00000101 and 11001000 = 00000000
and M,y   ; = and 5180 : 00000101 and 11010010 = 00000000
and (Z,x) ; = and 4110 : 00000101 and 11011100 = 00000100
and (Z),y ; = and 5180 : 00000101 and 11010010 = 00000000
```

O R A : 1 or 1 = 1; 1 or 0 = 1; 0 or 1 = 1; 0 or 0 = 0

```
Flags: Z,N
ora #B    ; ora #2 :     00000101 or 00000010 = 00000111
ora Z     ; ora 20 :     00000101 or 00001010 = 00001111
ora Z,x   ; = ora 60 :   00000101 or 00001110 = 00001111
ora M     ; = ora #255:  00000101 or 11111111 = 11111111
ora M,x   ; = ora 5170:  00000101 or 11001000 = 11001101
ora M,y   ; = ora 5180:  00000101 or 11010010 = 11010111
ora (Z,x) ; = ora 4110:  00000101 or 11011100 = 11011101
ora (Z),y ; = ora 5180:  00000101 or 11010010 = 11010111
```

EOR : 1 eor 1 = 0; 1 eor 0 = 1; 0 eor 1 = 1; 0 eor 0 = 0

```
Flags: Z,N
eor #2     ; eor #2:     00000101 eor 00000010 = 00000111
eor Z      ; eor 20:     00000101 eor 00001010 = 00001111
eor M      ; eor #255:   00000101 eor 11111111 = 11111010
eor Z,x    ; = eor 60:   00000101 eor 00001110 = 00001011
eor M,x    ; = eor 5170: 00000101 eor 11001000 = 11001101
eor (Z,x)  ; = eor 4110: 00000101 eor 11011100 = 11011001
eor (Z),y  ; = eor 5180: 00000101 eor 11010010 = 11010111
```

B I T : wie AND, aber <A> bleibt erhalten

```
Flags: Z,N,V; <v>=Bit 6 und <n>=Bit 7 aus adressierter Adresse
bit Z      ; bit 20:   00000101 and 00001010 = 0 ---> <z>=1;<v>=0
bit M      ; bit 5130: 00000101 and 11111111 = 5 ---> <z>=0;<v>=0
```

12.4.5 Verschiebe-Befehle (bitweise)

Diese Befehle beeinflussen nur den angesprochenen Speicherinhalt.

A S L : alle Bits um eins nach links, Bit 7 nach <c>,
Bit 0 mit 0 auffüllen

```
Flags: C,Z,N
asl        ; ?asl<A>:  00000101 ---> 00001010; <c>=0
asl Z      ; asl 20:   00001010 ---> 00010100; <c>=0
asl M      ; asl 5130: 11111111 ---> 11111110; <c>=1
asl Z,x    ; asl 60:   00001110 ---> 00011100; <c>=0
asl M,x    ; asl 5170: 11001000 ---> 10010000; <c>=1
```

L S R : alle Bits um 1 nach rechts, Bit 0 nach <c>,
Bit 7 mit 0 auffüllen

```
Flags: C,Z,N
lsr        ; ?lsr<A>:  00000101 ---> 00000010; <c>=1
lsr Z      ; lsr 20:   00001010 ---> 00000101; <c>=0
lsr M      ; lsr 5130: 11111111 ---> 01111111; <c>=1
lsr Z,x    ; lsr 60:   00001110 ---> 00000111; <c>=0
lsr M,x    ; lsr 5170: 11001000 ---> 01100100; <c>=0
```

R O L : Rotation mit C-Flag nach links

```
Flags: C,Z,N
rol       ; ?rol<A>:  00000101 / <c>=0 ---> 00001010 / <c>=0
rol Z     ; rol 20:   00001010 / <c>=0 ---> 00010100 / <c>=0
rol M     ; rol 5130: 11111111 / <c>=0 ---> 11111110 / <c>=1
rol Z,x   ; rol 60:   00001110 / <c>=1 ---> 00011101 / <c>=0
rol M,x   ; rol 5170: 11001000 / <c>=0 ---> 10010000 / <c>=1
```

R O R : Rotation mit C-Flag nach rechts

```
Flags: C,Z,N
ror       ; ?ror<A>:  00000101 / <c>=0 ---> 00000010 / <c>=1
ror Z     ; ror 20:   00001010 / <c>=0 ---> 00000101 / <c>=0
ror M     ; ror 5130: 11111111 / <c>=0 ---> 01111111 / <c>=1
ror Z,x   ; ror 60:   00001110 / <c>=1 ---> 10000111 / <c>=0
ror M,x   ; ror 5170: 11001000 / <c>=0 ---> 01100100 / <c>=0
```

12.4.6 Vergleichsbefehle

Vergleiche werden zwischen dem Register und dem angesprochenen Byte durchgeführt.
Das jeweilige Register bleibt unverändert erhalten.

C M P : Vergleich Byte und <A>

```
Flags: C,Z,N
cmp #B    ; cmp #2:   5 > 2   ---> <c>=1;<n>=0;<z>=0
cmp Z     ; cmp #20:  5 < 20  ---> <c>=0;<n>=1;<z>=0
cmp M     ; cmp 5130: 5 < 255 ---> <c>=0;<n>=0;<z>=0
cmp Z,x   ; cmp 60:   5 < 14  ---> <c>=0;<n>=1;<z>=0
cmp M,x   ; cmp 5170: 5 < 200 ---> <c>=0;<n>=0;<z>=0
cmp M,y   ; cmp 5180: 5 < 210 ---> <c>=0;<n>=0;<z>=0
cmp (Z,x) ; cmp 4110: 5 < 220 ---> <c>=0;<n>=0;<z>=0
cmp (Z),y ; cmp 5180: 5 < 210 ---> <c>=0;<n>=0;<z>=0
```

C P X : Vergleich Byte und <X>

```
Flags: C,Z,N
cpx #B    ; cpx #2:   40 > 2   ---> <c>=1;<n>=0;<z>=0
```

```
cpx Z      ; cpx 20:   40 > 10  ---> <c>=1;<n>=0;<z>=0
cpx M      ; cpx 5130  40 < 255 ---> <c>=0;<n>=0;<z>=0
```

C P Y : Vergleich Byte und <Y>

```
Flags: C,Z,N
cpy #2     ; cpy #2:   50 > 2   ---> <c>=1;<n>=0;<z>=0
cpy Z      ; cpy 20:   50 > 10  ---> <c>=1;<n>=0;<z>=0
cpy M      ; cpy 5130: 50 < 255 ---> <c>=0;<n>=0;<z>=0
```

12.4.7 Transportbefehle zwischen den Registern

T A X : <A> nach <X>

```
Flags: Z,N
tax        ; <X>=5; <A>=5
```

T A Y : <A> nach <Y>

```
Flags: Z,N
tay        ; <Y>=5; <A>=5
```

T X A : <X> nach <A>

```
Flags: Z,N
txa        ; <A>=40; <X>=40
```

T Y A : <Y> nach <A>

```
Flags: Z,N
tya        ; <A>=50; <Y>=50
```

T S X : Stackpointer <S> nach <X>

```
Flags: Z,N
tsx        ; <X>=255 (Beispiel für leeren Stack)
```

T X S : <X> nach Stackpointer (S)

```
txs        ; <S>=40
```

12.4.8 Sprungbefehle

Die relativen Sprungbefehle beginnend mit "B.." (für branch) erlauben Sprünge von maximal 128 Adressen vor oder zurück.

B E Q : Verzweige bei Z-Flag=1

B N E : Verzweige bei Z-Flag=0

B C C : Verzweige bei C-Flag=0
nach Vergleichsbefehlen: kleiner als

B C S : Verzweige bei C-Flag=1
nach Vergleichsbefehlen: größer als/ gleich

B M I : Verzweige bei N-Flag=1
Bit 7 bei letzter Operation gesetzt

B P L : Verzweige bei N-Flag=0
Bit 7 bei letzter Operation nicht gesetzt

B V C : Verzweige bei V-Flag=0

B V S : Verzweige bei V-Flag=1

J S R : Sprung zu Unterprogramm
Rückkehr nach RTS-Befehl

J M P : abslouter Sprung an beliebige Adresse

```
jmp M     ; jmp 5130 ---> bei (5130) im Programmlauf weiterfahren
                          direkter Sprung (jump)
jmp (Z)   ; jmp (20) = jmp 4110 ---> Sprung an die Adresse, die
            indirekt durch <20/21> mit LO/HI angegeben ist.
```

R T S : Rücksprung von einem Unterprogramm
(Programmende)

R T I : Rücksprung von Interrupt-Routine

B R K : Sprung zu der Adresse, die im BRK-Vektor steht

12.4.9 Beeinflussung der Flags des Statusregiters

C L C : C-Flag löschen

```
clc       ; ---> <c>=0
```

S E C : C-Flag setzen

```
sec       ; ---> <c>=1
```

C L D : Dezimalflag löschen

S E D : Dezimalflag setzen

C L V : Overflow-Flag (V) löschen

S E I : Interrupt-Flag setzen

```
sei       ; ---> kein Interrupt über IRQ-Vektor möglich
```

C L I : Interruptflag löschen

```
cli       ; ---> Interrupt wird ausgeführt über IRQ-Vektor
```

12.4.10 Lückenfüller

N O P : keine Operation - weiter bei nächster Adresse

13

ROM-Routinen – thematisch, mit Kurzbeschreibung

13 ROM-Routinen mit Kurzbeschreibung

Label	C64	40/80XX
Arithmetik		
ADD0.5	47177=$b849 FAC1 + 0.5 ---> FAC1	51583=$c97f
FMAL10	47842=$bae2 FAC1 mal 10 ---> FAC1	52248=$cc18
FDIV10	47870=$bafe FAC1 / 10 ---> FAC1	52276=$cc34
FACMIN	49076=$bfb4 -FAC1 ---> FAC1	53579=$d14b
FACABS	48216=$bc58 abs(FAC1) ---> FAC1	53622=$cd8e
ADD	47210=$b86a FAC2 + FAC1 ---> FAC1	51616=$c9a0
M-ADD	47207=$b867 MEMORY (A/Y) + FAC1 ---> FAC1	51613=$c9a0
SUB	47187=$b853 FAC2 - FAC1 ---> FAC1	51593=$c989
M-SUB	47184=$b850 MEMORY (A/Y) - FAC1 ---> FAC1	51590=$c986
INTMUL	45900=$b34c <113/114>=<$71/72> F1 mal MEMORY (95/96)=($5f/60) F2AD Produktwert ---> (X/A)	50295=$c477 <110/111>=<$6e/6f> (92/93)=($5c/5d)
MULT	47659=$ba2b FAC2 mal FAC1 ---> FAC1	52065=$cb61

Label	C64	40/80XX
M-MULT	47656=$ba28 MEMORY (A/Y) mal FAC1 ---> FAC1	52062=$cb5e
DIV	47890=$bb12 FAC 2 / FAC 1 ---> FAC 1	52296=$cc48
M-DIV	47887=$bb0f MEMORY (A/Y) / FAC1 ---> FAC1	52293=$cc45
SQRFAC	49009=$bf71 Quadratwurzel aus FAC1 ---> FAC1	53512=$d108
POTRAD	49019=$bf7b FAC2 hoch FAC1 ---> FAC1	53522=$d112
M-POT	49016=$bf78 MEMORY (A/Y) hoch FAC1 ---> FAC1	53519=$d10f
LOGNAT	47594=$b9ea ln(FAC1) ---> FAC1	52000=$cb20
EHOCHF	49133=$bfed e hoch FAC1 ---> FAC1	53636=$d184
SINUS	57963=$e26b sin(FAC1) ---> FAC1 (Bogenmaß RAD)	53897=$d289
COSIN	57956=$e264 cos(FAC1) ---> FAC1 (Bogenmaß RAD)	53890=$d282
TANG	58036=$e2b4 tan(FAC1) ---> FAC1 (Bogenmaß RAD)	53970=$d2d2
ARCTAN	58126=$e30e arctan(FAC1) ---> FAC1 (Bogenmaß RAD)	54060=$d3c2
POLNOM	57433=$e059 Polynomwert aus Konstantentabelle ab (A/Y) ---> FAC 1. Byte der Tabelle = Polynomgrad n folgende Bytes enthalten die Koeffizienten a_n bis a_0 als reelle Zahlen in 5-er Gruppen	53741=$d1ed

Label	C64	40/80XX
CMPFAC	48219=$bc5b vergleicht FAC1 mit MEMORY (A/Y) FAC < MEM ---> <A>:=255 FAC > MEM ---> <A>:= 1 FAC = MEM ---> <A>:= 0	52625=$cd91
SGNFAC	48171=$bc2b Vorzeichen von FAC1 ---> (A) "+" ---> <A>:= 1 "-" ---> <A>:=255 <FAC1> = 0 ---> <A>:=0	52577=$cd61
MEMFAC	48034=$bba2 reelle Zahl aus MEMORY (A/Y) ---> FAC1	52440=$ccd8
MEMFC2	47756=$ba8c reelle Zahl aus MEMORY (A/Y) ---> FAC2	52162=$cbc2
FACMEM	48087=$bbd7 FAC1 (reell) ---> MEMORY (X/Y)-Anfangsadresse	52493=$cd0d
FAC1/2	48143=$bc0f <FAC1> ---> FAC2	52549=$cd45
FAC2/1	48124=$bbfc <FAC2> ---> FAC1	52530=$cd32
ZUFALL	57495=$e097 Zufallszahl aus 0 bis 1 ---> FAC1, abhängig von der Zeit bzw. von <A>	53801=$d229 53804=$d22c

Umwandlungen

INTFLP	45969=$b391 positive/negative Integerzahl aus (Y/A) nach FAC1 Bereich: +32767/-32768	50364=$c4bc
ADRFLP	48201=$bc49 <98=$62> Integerzahl HI <99=$63> Integerzahl LO	52607=$cd7f <95=$5f> <96=$60>

Label	C64	40/80XX
	<X>=144; SEC; positive Integerzahl nach FAC1 Bereich: 0 bis 65535 (Adressbereich)	
FLPINT	47095=$b7f7	51501=$c92d
	<FAC 1>(reell)---> <FAC 1>(integer),LOW/HI=(Y/A)	
FLPSTR	48605=$bddd	53139=$cf93
	<FAC 1> ---> Ziffernfolge ab (256), Ende=Byte 0	
STRFAC	47029=$b7b5	51435=$c8eb
mit STRADR	<34=$22/35=$23> Stringanfang <31=$1f/32=$20>	
	<A>=Stringlänge; Ergebnis in FAC1	
BYTHEX	55098=$d73a	---
	<A>(Byte) ---> (A)(Hex-Byte)	
HEXBYT	55181=$d78d	---
	<A>(ASC-Code) ---> (A)(Byte)	
Bildschirm-Ausgaben		
CHROUT(BSOUT)	65490=$ffd2 KERNAL	65490=$ffd2
	<A> auf den Bildschirm als ASCII-Zeichen	
INTOUT	48589=$bdcd	53123=$cf83
	Die Integerzahl wird direkt aus (X/A) mit LO/HI auf den Bildschirm gebracht.	
CURPOS	58732=$e56c	---
	berechnet mit CURZEI und CURSPA Cursorposition	
FPOUTX	48599=$bdd7	53133=$cf8d
	C64: LDY #1 / JSR 48599	53133=$cf8d
FLPOUT/CR	43708=$aabc	---
	C64: Ausgabe <FAC> mit anschließendem CR	
FLPSTR	48605=$bddd	53139=$cf93
	<FAC> ---> String ab (256)	
STROUT	43813=$ab25	47908=$bb24
	<X>=Stringlänge	
mit STRADR	<34/35> Stringanfang	<31/32>

Label	C64	40/80XX
STR-O	43806=$able	47901=$bbld
	<A/Y>=Stringadresse; Ende des Strings: Byte 0	
BYTOUT	---	55074=$d722
	<A> ---> Hex ---> Schirm	
ADROUT	---	55063=$d717
	<251/252> ---> Hex ---> Schirm	
OUT2	---	55089=$d731
	<X> ---> Schirm, <A> ---> Schirm	

Adressen

CURZEI	214=$d6	216=$d8
CURSPA	211=$d3	198=$c6
ZEIPTR	209/210=$d1	196/197=$c4/c5

Eingabe-Routinen

GETIN	65508=$ffe4 KERNAL	65508=$ffe4
	1 Zeichen ---> (A)	
BASIN	65487=$ffcf KERNAL	65487=$ffcf
	String ---> Schirm ---> einzeln nach (A)	
INLINE	42336=$a560	46306=$b4e2
	Zeile nach (512)...	
	nimmt bis zu 80 Zeichen auf	
HEXINB	---	55139=$d767
	Byte in Hexeingabe ---> (A)	
HEXINA	---	55124=$d754
	Adresse(vierstellig) in Hexform ---> (251/252)	

Label	C64	40/80XX
Variablen-Verwaltung		
PTRVAR	45287=$b0e7 sucht Variable mit 1.Name/2.Name=<VARNAM>	49543=$c187
VARNAM1/2	(69/70)=($45/46) Variablenamel/2 für PTRVAR	(66/67)=($42/43)
VARADR	(71/72)=($47/48) dort steht die Anfangsadresse LO/HI der Variablen nach Aufruf von PTRVAR	(68/69)=($44/45)
TXTTAB	(43/44)=($2b/2c) Zeiger auf Beginn des BASIC-Textes	(40/41)=($28/29)
VARTAB	(45/46)=($2d/2e) Zeiger auf Beginn der einfachen Variablen	(42/43)=($3a/3b)
ARYTAB	(47/48)=$(2f/30) Zeiger auf Beginn der indizierten Variablen (Felder=Arrays)	(44/45)=($3c/3d)
VAREND	(49/50)=($31/32) Zeiger auf Ende der gesamten Variablentabelle	(46/47)=($3e/3f)
MAXMEM	(55/56)=($37/38) Zeiger auf RAM-Obergrenze	(52/53)=($34/35)
Ein- und Ausgabe-Routinen		
OPEN	62282=$f34a öffnet eine Datei mit LA,GA,SA auf ein Gerät bei Floppy-Dateien sind NAMAD und NAMLEN notwendig	62819=$f563
BFOUT	--- gibt ähnlich wie OPEN einen Befehlsstring aus zugeordnet **LA, GA, SA**	55963=$da9b
LOAD	--- lädt Datei oder Programm, das mit OPEN geöffnet wurde, dazu vorher <STATUS>=0, <LOVE>=0 setzen	62472=$f408

Label	C64	40/80XX
	Programmzeiger 'Anfang' und 'Ende' werden gesetzt	
LOADXX	62648=$f4b8 wie LOAD, aber ohne Veränderung der Zeiger	62294=$f356
SUFTAB	62223=$f30f stellt für eine bereits geöffnete Datei GN,SA,LA bereit aus der intern geführten Tabelle Vorbereitung: <LA> ---> (A)	62145=$f2c1
SETTAB	62239=$f31f setzt die mit SUFTAB gefundenen Parameter in die vorgesehenen Zeropage-Adressen	62157=$f2cd
CLOSEA	62097=$f291 schließt eine noch offene Datei Vorbereitung: logische Adresse nach (LA) bringen	62178=$f2e2
CLOSEL	--- schließt Datei mit LA aus (210)=**LA**	62176=$f2e0
TWAIT	63213=$f6ed verhindert vorzeitigen Rücksprung, wenn die Zentraleinheit schneller als das Peripheriegerät ist	63787=$f92b
TALK	60681=$ed09 aktiviert das mit OPEN angesprochene Gerät als Sender (Talker) Vorbereitung: <STATUS>=0	61650=$f0d2
UNTALK	60911=$edef versetzt den (IEC-)Bus nach TALK wieder in neutrale Zustand, Gerät wird als Talker deaktiviert	61878=$f1b6
LISTEN	60684=$ed0c aktiviert das angesprochene Gerät als Empfänger Vorbereitung: OPEN, <STATUS>=0	61653=$f0d5
UNLISN	60926=$edfe versetzt den (IEC-)Bus nach Tätigkeit als Listener wieder in neutralen Zustand	61881=$f1b9

Label	C64	40/80XX
GETSA	---	55599=$d92f
	nächste freie Sekundäradresse holen ---> SA	
SASENL	60857=$edb9	61763=$f143
SASENT	60871=$edc7	61763=$f143
	sendet die Sekundäradresse und bereitet damit einen Kanal für die folgende Ein- oder Ausgabe vor. Dieser Befehl muß also immer dem ersten INBUS oder dem ersten OUTBUS vorangehen. Vorbereitung: <A>=<SA>, also Akku mit Sekundäradresse belegen, die man ebentuell aus (SA) holt	
BSOUT(CHROUT)	65490=$ffd2 KERNAL	65490=$ffd2
OUTBUS	60893=$eddd	61854=$f19e
	sendet das im (A)-Register befindliche Byte über den IEC-Bus auf den aktiven Kanal Vorbereitung: OPEN, LISTEN, SASEND, <A>	
BASIN(CHRIN)	65487=$ffcf KERNAL	65487=$ffcf
	holt ein Zeichen nach (A) über aktiven Kanal. Bei Standard-I/O: mit Bildschirmausgabe bis CR anschließend erstes Zeichen in (A).	
INBUS	60947=$eel3	61888=$f1c0
	holt über den (IEC-)Bus ein Byte nach (A) vom aktiven Kanal Vorbereitung: OPEN, TALK, SASEND	
CHKOUT	65481=$ffc9 KERNAL	65481=$ffc9
	leitet Ausgabe über den (IEC-)Bus auf das angesprochene Gerät um (statt Schirm) Vorbereitung: OPEN, <LA> ---> (X)	
CHKIN	65478=$ffc6 KERNAL	65478=$ffc6
	Eingabe über den (IEC-)Bus vom angesprochenen Gerät (statt Tastatur) Vorbereitung: OPEN, <X>=<LA>	
CLRCH	65484=$ffcc KERNAL	65484=$ffcc
	schaltet als Eingabegerät die Tastatur (GA=0) und als Ausgabegerät den Bildschirm (GA=3) schließt alle Kanäle, aber keine Dateien	

Label	C64	40/80XX
CLALL	65511=$ffe7 KERNAL	65511=$ffe7
	schließt alle Kanäle und Dateien	

Adressen zu den Ein/Ausgabe-Routinen

LA	184=$b8	210=$d2
	logische Adresse zur Bezeichnung der Datei	
SA	185=$b9	211=$d3
	Sekundäradresse zur Kanalbereitstellung	
GA	186=$ba	212=$d4
	Gerätenummer für angesprochenes Gerät	
NAMADR	187/188=$bb/bc	218/219=$da/db
	LO/HI der Anfangsadresse des Dateinamens oder eines Befehlsstrings	
NAMLEN	183=$b7	209=$d1
	Adresse zur Bereitstellung der Länge des Namens	
STATUS	144=$90	150=$96
	Byte für Fehler- bzw. Ende-Erkennung der Datei kein Fehler: <STATUS>=0	
LVFLAG	147=$93	157=$9d
	Load- bzw. Verify-Flag. Laden: <LVFLAG>=0	

BASIC-Text-Routinen und -adressen

CHRGET	00115=$0073	00112=$0070
	erhöht Programmzeiger, BASIC-Textzeichen ---> (A)	
CHRGOT	00121=$0079	00118=$0076
	beläßt Programmzeiger, holt noch einmal das letzte BASIC-Textzeichen nach (A)	
GETBYT	47003=$b79b	51409=$c8d1
	BASIC-Text von (Programmzeiger +1) bis Trennzeichen als Byte-Wert ---> (X)	

Label	C64	40/80XX
VALBYT	47006=$b79e BASIC-Text von Programmzeiger bis Trennzeichen als Byte-Wert ---> (X)	51412=$c8d4
VAREAL	44426=$ad8a BASIC-Text wird als reelle Zahl gemäß den Rechenregeln berechnet bis zum Trennzeichen ---> FAC1	48516=$bd84
VALINT	45493=$b1b5 BASIC-Text wird bis Trennzeichen als Ganzzahl ausgewertet <FAC1+3/FAC1+4>=HI/LO	49888=$c2e0
INTADR	43371=$a96b <20/21>=LO/HI	47350=$b8f6 <17/18>=LO/HI
VALPAR	44446=$ad9e wertet Zeichenfolge bis Trennzeichen als Zahl oder String aus: Zahl ---> FAC1, <PARFLG>=0, <TYPFLG>=128(integer) oder <TYPFLG>=0(reell) String: Zeiger auf Deskriptor = <FAC1+3/FAC1+4> <PARFLG>=255	48536=$bd98
VALSTR	46755=$b6a3 Aufruf nach VALPAR <34/35> Stringanfangsadresse <31/32> Stringlänge = <A>	51152=$c7b5
VALKLA	44785=$aef1 wertet Zeichenfolge zwischen "(" und ")" aus wie VALPAR	48873=$bee9
GETVAR	45195=$b08b liest Zeichenfolge als Variablenname, sucht sie oder legt sie an in <A/Y> und **VARADR** steht Variablenadresse	49451=$c12b
GETFIP	57812=$eld4 holt Dateinamen mit eventuell vorangestellter Laufwerknummer, Gerätenummer und eventuelle Sekundäradresse; setzt diese Fileparamater in die Zeropage	62589=f47d

Label	C64	40/80XX
STRPLZ	46197=$b475	50590=$c59e
	mit <A>=Stringlänge wird entsprechend neuer Platz im Stringbereich "angehängt". <FAC1+1/FAC1+2>=Stringanfangsadresse	
BPRINT	43982=$aaa2	47805=$babd
	druckt Zeichenfolge ab PRGPTR als String aus, dazu PRGPTR auf Anfang des auszugebenden Ausdrucks	

Trennzeichen - Syntax-Routinen

KLMAUF	44794=$aefa	48882=$bef2
	<A>=40 ---> kein Syntaxfehler, Programmzeiger erhöhen und nächstes Zeichen ---> (A) BASIC-Zeilennummer LO/HI	
KLAMZU	44791=$aef7	48479=$beef
	<A>=41 (Klammer zu)? - sonst wie oben	
KOMMA	44797=$aefd	48885=$bef5
	<A>=44 (Komma ?) - sonst wie oben	
PRFZEI	44799=$aeff	48887=$bef7
	<A> beliebig - sonst wie oben BASIC-Text von (Programmzeiger +1) bis Trennzeichen	
STRTYP	44431=$ad8f	48521=$bd89
	<PARFLG>=255 ---> keine Fehlermeldung, also liegt ein String vor (nach der Auswertung mit VALPAR oder VALSTR o.ä.)	
NUMTYP	44428=$ad8c	48519=$bd87
	<PARFLG>=0 ---> keine Fehlermeldung, also liegt eine numerische Variable nach der Auswertung des BASIC-Textes vor	
ERRXX	42082=$a462	46048=$b3e0
	druckt Fehlermeldung	

Label	C64	40/80XX
ERROR	42042=$a43a druckt Fehlermeldung, dazu Fehlernummer in (X) Alle Kanäle werden geschlossen, nicht jedoch die eventuell offenen Dateien.	46031=$b3cf Offset

Adressen für BASIC-Text-Routinen

FAC1 (FAC)	97...102=$61...66	94...99=$5e...63
FAC+1/2	98/99=$62/63	95/96=$5f/60
FAC+3/4	100/101=$64/65	97/98=$61/62
STRADR	34/35=$22/23 Adresse des Stringanfangs nach VALSTR	31/32=$1f/20
VARADR	71/72=$47/48 Adresse des Variablenanfangs LO/HI nach GETVAR	68/69=$44/45
LINNUM	20/21=$14/15 BASIC-Zeilennummer LO/HI	17/18=$11/12
PARFLG	13=$0d	07=$07
TYPFLG	14=$0e	08=$08

Diverse ROM-Routinen

KALTPT	43130=$a87a setzt BASIC-Zeiger für einen Kaltstart	47114=$b80a
WARMPT	42638=$a68e setzt BASIC-Zeiger für einen Warmstart	46626=$b622
MRUN	42948=$a7c4 startet BASIC-Programm, Zeiger vorher setzen	46943=$b75f
MGOTO	43171=$a8a3 bricht Programm ab, startet bei Zeilennummer BASLIN Zeilennummer auch nach LINNUM setzen	47155=$b833

Label	C64	40/80XX
BLINAD	42515=$a613 sucht Adresse der Zeilennummer in LINNUM, setzt sie in LINADR mit LO/HI	46499=$b5a3
TEXMOD	--- schaltet auf Textmodus um (Zwischenräume)	57368=$e018
GRAMOD	--- schaltet auf Graphikmodus um (ohne Zwischenräume)	57371=$e01b
STOPRY	65505=$ffe1 KERNAL Programmabbruch, READY-Modus	65505=$ffe1
STOPO	63213=$f6ed STOP-Taste gedrückt ---> Z-Flag gesetzt	62261=$f335
MREADY	42100=$a474 beendet Maschinenprogramm, READY	46079=$b3ff
TRABLO	41919=$a3bf verschiebt Datenblock von BABL bis EABL mit neuem Ende auf ENBL	45911=$b357
TRPSET	64398=$fb8e setzt **LDPTR** als Transportzeiger zum Abspeichern	64443=$fbbb
DATOUT	63017=$f629 gibt mit Transportzeiger Daten auf (IEC)-Bus von **PRGANF** bis **PRGEND** aus	63260=$f71c
DIRPR	62895=$f5af prüft vor einer Standard-Meldung auf Direktmodus	62281=$f349
IRQ	59953=$ea31 Standard-Interrupt-Einsprung	58453=$e455

Adressen

Label	C64	40/80XX
BASLIN	57/58=$39/3a laufende BASIC-Zeile	54/55=$36/37
LINNUM	20/21=$14/15 Zeilennummer LO/HI	17/18=$11/12
LINAD	95/96=$5f/60 Ergebnis von **BLINAD** (Zeilennummer LO/HI)	92/93=$5c/5d
BABL	95/96=$5f/60 Blockverschiebezeiger "Beginn des alten Blocks"	92/93=$5c/5d
EABL1	90/91=$5a/5b Blockverschiebezeiger "Ende des alten Blocks +1"	87/88=$57/58
ENBL1	88/89=$58/59 Blockverschiebezeiger "Ende des neuen Blocks +1"	85/86=$55/56
BEGDAT	193/194=$c1/c2 Datenanfangsadresse LO/HI	251/252=fb/fc
ENDDAT	174/175=$ae/af Datenendadresse LO/HI	201/202=$c9/ca
LDPTR	172/173=$ac/ad Ladezeiger für Routine **DATOUT**	199/200=$c7/c8
PCR	53272=$d018 Peripherie-Control-Register <PCR>=12 ---> Textzeichen, <PCR>=14 ---> Graphik	59468=$e84c
IRQVEC	788/789=$0314/0315 Interrupt-Vektor	144/145=$90/91
NMIVEC	792/793=$0318/0319 NMI-Vektor (Einsprung nach NMI-Signal)	148/149=$94/95
BRKVEC	790/791=$0316/0317 BRK-Vektor (Einspung nach BRK-Befehl)	146/147=$92/93

14

ROM-Routinen – alphabetisch

Label	C64		40/80XX	Label
ADD	47210=$b86a		51616=$c9a0	ADD
ADDO.5	47177=$b849		51583=$c97f	ADDO.5
ADRFLP	48201=$bc49		52607=$cd7f	ADRFLP
ADROUT	---		55063=$d717	ADROUT
ARCTAN	58126=$e30e		54060=$d3c2	ARCTAN
BASIN	65487=$ffcf		65487=$ffcf	BASIN
BFOUT	---		55963=$da9b	BFOUT
BLINAD	42515=$a613		46499=$b5a3	BLINAD
BPRINT	43682=$aaa2		47805=$babd	BPRINT
BSOUT	65490=$ffd2		65490=$ffd2	BSOUT
BYTHEX	---		55098=$d73a	BYTHEX
BYTOUT	---		55074=$d722	BYTOUT
CHKIN	65478=$ffc6	KERNAL	65478=$ffc6	CHKIN
CHKOUT	65481=$ffc9	KERNAL	65481=$ffc9	CHKOUT
CHRGET	115=$73		112=$70	CHRGET
CHRGOT	121=$0079		118=$0076	CHRGOT
CHROUT	65490=$ffd2		65490=$ffd2	CHROUT
CLALL	65511=$ffe7	KERNAL	65511=$ffe7	CLALL
CLOSE	65475=$ffc3	KERNAL	65475=$ffc3	CLOSE
CLOSEA	62097=$f291		62178=$f2e2	CLOSEA
CLOSEL	---		62176=$f2e0	CLOSEL
CLRCH	65484=$ffcc	KERNAL	65484=$ffcc	CLRCH
CMPFAC	48219=$bc5b		52625=$cd91	CMPFAC
COSIN	57956=$e264		53890=$d282	COSIN
CURPOS	58732=$e56c		---	CURPOS
DATOUT	63017=$f629		63260=$f71c	DATOUT
DIRPR	62895=$f5af		62281=$f349	DIRPR
DIV	47890=$bb12		52296=$cc48	DIV
EHOCHF	49133=$bfed		53636=$d184	EHOCHF
ERR80	---		62895=$f5af	ERR80
ERROR	42042=$a43a		46031=$b3cf	ERROR
ERRXX	42082=$a462		46048=$b3e0	ERRXX
FAC1/2	48143=$bc0f		52549=$cd45	FAC1/2
FAC2/1	48124=$bbfc		52530=$cd32	FAC2/1
FACABS	48216=$bc58		53622=$cd8e	FACABS

Label	C64	40/80XX	Label
FACMEM	48087=$bbd7	52493=$cd0d	FACMEM
FACMIN	49076=$bfb4	53579=$d14b	FACMIN
FDIV10	47870=$bafe	52276=$cc34	FDIV10
FLPINT	47095=$b7f7	51501=$c92d	FLPINT
FLPOUT	43708=$aabc	53133=$cf8d	FLPOUT
FLPSTR	48605=$bddd	53139=$cf93	FLPSTR
FMALL0	47842=$bae2	52248=$cc18	FMALL0
FPOUTX	48599=$bdd7	---	FPOUTX
GETBYT	47003=$b79b	51409=$c8d1	GETBYT
GETFIP	57812=$e1d4	62589=$f47d	GETFIP
GETIN	65508=$ffe4	65508=$ffe4	GETIN
GETSA	---	55599=$d92f	GETSA
GETVAR	45195=$b08b	49451=$c12b	GETVAR
GRAMOD	---	57371=$e01b	GRAMOD
HEXBYT	---	55181=$d78d	HEXBYT
HEXINA	---	55124=$d754	HEXINA
HEXINB	---	55139=$d767	HEXINB
INBUS	60947=$ee13	61888=$f1c0	INBUS
INLINE	42336=$a560	46306=$b4e2	INLINE
INTADR	43371=$a96b	47350=$b8f6	INTADR
INTFLP	45969=$b391	50364=$c4bc	INTFLP
INTMUL	45900=$b34c	50295=$c477	INTMUL
INTOUT	48589=$bdcd	53123=$cf83	INTOUT
IRQ	59953=$ea31	58453=$e455	IRQ
IRVEC-LO	788=$0314	144=$90	IRVEC-LO
IRVEQ-HI	799=$0315	145=$91	IRVEQ-HI
KALTPT	43130=$a87a	47114=$b80a	KALTPT
KLAMZU	44791=$aef7	48479=$beef	KLAMZU
KLMAUF	44794=$aefa	48882=$bef2	KLMAUF
KOMMA	44797=$aefd	48885=$bef5	KOMMA
LISTEN	60684=$ed0c	61653=$f0d5	LISTEN
LOAD	---	62472=$f408	LOAD
LOADXX	62648=$f4b8	62294=$f356	LOADXX
LOGNAT	47594=$b9ea	52000=$cb20	LOGNAT
M-ADD	47207=$b867	51613=$c9a0	M-ADD

Label	C64		40/80XX	Label
M-DIV	47887=$bb0f		52293=$cc45	M-DIV
M-MULT	47656=$ba28		52062=$cb5e	M-MULT
M-POT	49016=$bf78		53519=$d10f	M-POT
M-SUB	47184=$b850		51590=$c986	M-SUB
MEMFAC	48034=$bba2		52440=$ccd8	MEMFAC
MEMFC2	47756=$ba8c		52162=$cbc2	MEMFC2
MGOTO	43171=$a8a3		47155=$b833	MGOTO
MREADY	42100=$a474		46079=$b3ff	MREADY
MRUN	42948=$a7c4		46943=$b75f	MRUN
MULT	47659=$ba2b		52065=$cb61	MULT
NUMTYP	44428=$ad8c		48519=$bd87	NUMTYP
OPEN	62282=$f34a		62819=$f563	OPEN
OUT2	---		55089=$d731	OUT2
OUTBUS	60893=$eddd		61854=$f19e	OUTBUS
PGRPTR	122/123=$7a/7b		119/120=$77/78	PGRPTR
POLNOM	57433=$e059		53741=$d1ed	POLNOM
POTRAD	49019=$bf7b		53522=$d112	POTRAD
PRFZEI	44799=$aeff		48887=$bef7	PRFZEI
PTRVAR	45287=$b0e7		49543=$c187	PTRVAR
SASEND	---		61763=$f143	SASEND
SASENL	60857=$edb9			SASENL
SASENT	60871=$edc7			SASENT
SETTAB	62239=$f31f		62157=$f2cd	SETTAB
SGNFAC	48171=$bc2b		52577=$cd61	SGNFAC
SINUS	57963=$e26b		53897=$d289	SINUS
SQRFAC	49009=$bf71		53512=$d108	SQRFAC
STOPO	63213=$f6ed		62261=$f335	STOPO
STOPRY	65505=$ffe1	KERNAL	65505=$ffe1	STOPRY
STR-O	43806=$ab1e		47901=$bb1d	STR-O
STRADR	34/35=$22/23		31/32=$1f/20	STRADR
STRFAC	47029=$b7bd		51435=$c8eb	STRFAC
STROUT	43813=$ab25		47908=$bb24	STROUT
STRPLZ	46197=$b475		50590=$c59e	STRPLZ
STRTYP	44431=$ad8f		48521=$bd89	STRTYP
SUB	47187=$b853		51593=$c989	SUB

Label	C64	40/80XX	Label
SUFTAB	62223=$f30f	62145=$f2c1	SUFTAB
TALK	60681=$ed09	61650=$f0d2	TALK
TANG	58036=$e2b4	53970=$d2d2	TANG
TEXMOD	---	57368=$e018	TEXMOD
TRABLO	41919=$a3bf	45911=$b357	TRABLO
TRPSET	64398=$fb8e	64443=$fbbb	TRPSET
TWAIT	63213=$f6ed	63787=$f92b	TWAIT
UNLISN	60926=$edfe	61881=$f1b9	UNLISN
UNTALK	60911=$edef	61878=$f1b6	UNTALK
VALBYT	47006=$b79e	51412=$c8d4	VALBYT
VALINT	45493=$b1b5	49888=$c2e0	VALINT
VALKLA	44785=$aef1	48873=$bee9	VALKLA
VALPAR	44446=$ad9e	48536=$bd98	VALPAR
VALSTR	46755=$b6a3	51152=$c7b5	VALSTR
VARADR	71/72=$47/48	68/69=$44/45	VARADR
VAREAL	44426=$ad8a	48516=$bd84	VAREAL
VARNAM	69/70=$45/46	66/67=$42/43	VARNAM
WARMPT	42638=$a68e	46626=$b622	WARMPT
ZUFALL	57495=$e097	53801=$d229	ZUFALL

15

Wichtige Adressen – alphabetisch

Label	C64		40/80XX	Label
ARYTAB	47/48=$2f/30		44/45=$3c/3d	ARYTAB
BABL	95/96=$5f/60		92/93=$5c/5d	BABL
BASLIN	57/58=$39/3a		54/55=$36/37	BASLIN
BEGDAT	193/194=$c1/c2		251/252=$fb/fc	BEGDAT
BRKVEC	790/791=$0316/0317		146/147=$92/93	BRKVEC
BSADHI	217=$d9		59246=$e76e	BSADHI
BSADLO	60656=$ecf0		59221=$e755	BSADLO
CURSPA	211=$d3		198=$c6	CURSPA
CURZEI	214=$d6		216=$d8	CURZEI
DEVIN	153=$99		175=$af	DEVIN
DEVOUT	154=$9a		176=$b0	DEVOUT
EABL1	90/91=$5a/5b		87/88=$57/58	EABL1
ENBL1	88/89=$58/59		85/86=$55/56	ENBL1
ENDDAT	174/175=$ae/af		201/202=$c9/ca	ENDDAT
EXPON	94=$5e		91=$5b	EXPON
F1 (intmul)	113/114=$71/72	LO/HI	110/111=$6e/6f	F1
F2AD	95/96=$5f/60		92/93=$5c/5d	F2AD
FAC1	97..101=$61..65		94..99=$5e/63	FAC1
FAC2	105..109=$69..6d		102..107=$66/6b	FAC2
GA	186=$ba		212=$d4	GA
IRQVEC	788/789=$0314/0315		144/145=$90/91	IRQVEC
KEY	203=$cb		151=$97	KEY
KEY(SK)			155=$9b	KEY(SK)
LA	184=$b8		210=$d2	LA
LDPTR	172/173=$ac/ad		199/200=$c7/c8	LDPTR
LINAD	95/96=$5f/60		92/93=$5c/5d	LINAD
LINNUM	20/21=$14/15		17/18=$11/12	LINNUM
LINNUM	57/58=$39/3a		17/17=$11/12	LINNUM
LVFLAG	147=$93		157=$9d	LVFLAG
MAXMEM	55/56=$37/38		52/53=$34/35	MAXMEM
NAMAD	187/188=$bb/bc		218/219=$d/db	NAMAD
NAMLEN	183=$b7		209=$d1	NAMLEN
NMIVEC	792/793=$0318/0319		148/149=$94/95	NMIVEC
PADAT	56320=$dc00 (CP2)		59471=$e8f4	PADAT
PADAT2	56321=$dc01 (CP1)			PADAT2

Label	C64	40/80XX	Label .
PARFLG	13=$0d	07=$07	PARFLG
PCR	53272=$d018	59468=$e84c	PCR
PGRPTR	122/123=$7a/7b	119/120=$77/78	PGRPTR
PORTA	56577=$dd01	59459=$e843	PORTA
Reelle Zahlen:			
RZ -0.5	224/185=$e0/b9	022/203=$16/cb	RZ -0.5
RZ 0.5	17/191=$11/bf	199/208=$c7/d0	RZ 0.5
RZ 1.00	188/185=$bc/b9	242/202=$f2/ca	RZ 1.00
RZ 0.25	234/226=$ea/e2	08/211=$08/d3	RZ 0.25
RZ 10	249/186=$f9/ba	047/204=$2f/cc	RZ 10
RZ 2pi	009/227=$09/e3	039/211=$27/d3	RZ 2pi
RZ pi	168/174=$a8/ae	160/190=$a0/be	RZ pi
RZ pi/2	224/226=$e0/e6	254/210=$fe/d2	RZ pi/2
RZ SQR(2)	219/185=$db/b9	017/203=$11/cb	RZ SQR(2)
SA	185=$b9	211=$d3	SA
SHIFLG	654=$028e	152=$98	SHIFLG
STATUS	144=$90	150=$96	STATUS
STRADR	34/35=$22/23	31/32=$1f/20	STRADR
TXTTAB	43/44=$2b/2c	40/41=$28/29	TXTTAB
TYPFLG	14=$0e	08=$08	TYPFLG
USRVEC	785/786=$0311/0312	01/02=$01/02	USRVEC
VARADR	71/72=$47/48	68/69=$44/45	VARADR
VAREND	49/50=$3b/3c	46/47=$3e/3f	VAREND
VARNAM	69/70=$45/46	66/67=$42/43	VARNAM
VARTAB	45/46=$2d/2e	42/43=$3a/3b	VARTAB
ZEIPTR	209/210=$d1/d2	196/197=$c4/c5	ZEIPTR

Adressen zur Spriteverwaltung (nur C64)

SP00 bis	2040=$07f8 bis	
SP07	2047=$07ff	
	Sprite-Pointer für Sprite 00 bis Sprite 07	
SPRAKT	53269=d015	
	Spriteadtivierungsregister (bitweise)	
COL00 bis	53287=$d027 bis	
COL07	53294=$d010	
	Farbregister der Sprites 00 bis 07	
MUCOL0	53285=$d025	Mehrfarbenregister 0
MUCOL1	53286=$d026	Mehrfarbenregister 1
MUCOLR	53276=$d01c	Mehrfarben-Steuer-Register
EXPHOR	53277=$d01d	Horizontal-Vergrößerungs-Register
EXPVER	53271=$d017	Vertikal-Vergrößerungs-Register
X00	53248=$d000 und	
Y00	53249=$d001	Sprite-Koordinaten x/y für SP00
...		
X07	53262=$d00e	
Y07	53263=$d00f	Sprite-Koordinaten x/y für SP07
X-MSB	53264=$d010	für x>255 Bit setzen
KOLLSS	53278=$d01e	Kollisionserkennung Sprites (Bits)
KOLLSD	53279=$d01f	Kollision Sprites/Daten (Bits)

16 Stichwortverzeichnis

ADC 51f
ADD 73f
Addieren 51,289
Adresse 20
ADRFLP 69
ADROUT 123f
AND 27,139,139,291
arithm. Verknüpfung 289ff
ASL 30,58f,292
ASSEMBLER 15f,286ff
Ausgabebefehle 288f

BASIC-Text 225ff
BASIC-Zeile 268
BASIC-Zeiger 157
BASIN 143ff
Befehlssatz 286ff
Befehlsstring 176ff
BEQ 28
Bewegungssimultaion 124ff
BIT 27
Bits 54
BLINAD 270
Blocktransport 272f
BNE 28
BSOUT 104f
Bus 163ff
BYTEHEX 68
BYTOUT 123

C-Flag 52,57,285
CHKIN 172f
CHKOUT 171f,177f
CHRGET 226f
CHRGOT 226f
CHROUT 92,104f,138ff
CLALL 167,172
CLI 46
CLOSE. 73,179
CLRCH 172,178
Cursorposition 105ff

Data 199f
Datei 171ff
DDRA 26
DEVIN 143
DEVOUT 209,210
Direktzugriff 216
Disassembler 15
Dividieren 74f
Drucker 168ff

EHOCHF 83f
Eingabebefehle 287f
EOR 54f,292
Exponent 114,120,154f
ERROR 235

FAC 61f,73
FACABS 79
FACMEM 64
FACMIN 78
FACSTR 109f
fastdisk 216ff
FDIV10 74f
Fehlermeldung 233ff
Flag 113,42f,285,296
Flagbeeinflussung 285,296
Floppy 205,208f,216
FLPINT 65f
FLPOUT 55,64,103,103f
FLPSTR 67
FMAL10 75

GA 163f
Geräteadresse 164ff
GETBYT 228f,232
GETFIP 274
GETIN 137ff
GETVAR 243f

Halbieren 58f
HEXBYT 68

HEXINA 148
HEXINB 148
Hexzahlen 18,21
HI 18,56
HIMEM 257f,257f

IEC-Bus 163f
INBUS 190
INLINE 145,145ff
INTADR 240
Integerzahlen 51ff,155f
Interpreter 225
Interrupt 41
INTFLP 55,63
INTMUL 60
INTOUT 53,103,52f,103
IRQ 41,256,269
IRQ-VEKTOR 41,45f,256

Joystick 25,26f

Kompaßanzeige 124ff
Konstante (ROM) 62f
Koordinaten 29
künstlicher Cursor 138ff

LA 163f
Label 20
Laden 196ff,275f,287
LISTEN 166
LO 18,56
LOADXX 197
Logartihmieren 83
log. Verknüpfungen 291,163ff
LOGNAT 83
LSR 59
LVFLAG 196f
Lückenfüller 296

M-ADD 77
M-DIV 80f
M-MULT 79
M-SUB 77f
Maschinensprache 15f
MEMFAC 63
MGOTO 267ff
Modul 254ff
Modulverknüpfung 254ff
MRUN 267
MSAVE 274
Multiplizieren 60f

N-Flag 52,285
negative Ganzzahlen 54ff

ONSTRING 244ff
OPEN 171
ORA 139,139,291
OUT2 124
OUTBUS 190

PADAT 26
PARFLAG 241f
PCR 271
Parameter 225f
Peripherie 163ff
Pixel 33
POLNOM 89ff
Polynom-Auswertung 89ff
positive Ganzzahlen 51ff
posload 276ff
possave 279ff
Potenzieren 81f,82
print-using 111ff
print at 233ff
printdirector 208ff
PTRVAR 158f

Quadratwurzel 81
quickdirektor 202ff

Realzahl-Addition 75ff
Rechnen 73ff,128f,289
Record 181ff
reelle Zahlen 154ff
Registerformat 61f,285,285
REL-Dateien 181ff
Richtungen 25f,125f

ROL 95,293,293
ROR 293

SA 163f
SASEN. 166
SBC 51f,56
SEI 46
Sekundäradresse 163ff
SEQ-Dateien 175
SETTAB 173
SHFLAG 268
Spalte 105f
Speicher 274f
Speicherformat 61,279f,288
Sprite 28ff,31,31,33
Sprungbefehle 295
Sprungleiste 258ff,180
STATUS 165
Statusregister 285
STOPO 272
STOPRY 272
STR-O 111,117
STRFAC 66,141
Stringvariable 156
STROUT 109
STRPLZ 247
Struktogramm 118
Subtrahieren 56,77,290
SUFTAB 173
Syntax 236ff,236ff

Takt 42,125,42ff,131f
TALK 165
Tastatur 137f
test onstring 253f
testtakt 44f
Trailer 246
Transport 294
TYPFLG 241f

UNTALK 165
USERPORT 170

VALBYT 231f,232
VALINT 239f
VALKLA 239
VALPAR 241f
VALSTR 243
VARADR 158f
VAREAL 238
Variable anlegen 153f
VARNAM 158f
Verdoppeln 58
Vergleich 94f,293f,293
Verschiebe-Befehl 292f,272f
Vorzeichenprüfung 96f

Warmstart 267
Warteschleife 29,42,34,42f
Winkelfunktionen 85ff
Wurzelziehen 81

Z-Flag 27f,52,285
Zahleneingabe 141f
Zeiger 46,157,203f,226ff
Zeropage 150f,247,269
Zufallszahl 84f

Weitere Fachbücher aus unserem Verlagsprogramm

COMMODORE 16/116

W. Besenthal/J. Muus

Alles über den C16

Juli 1986, 292 Seiten

Dieses Buch ist ein Lern- und Nachschlagewerk für jeden Commodore-Anwender. Es ist übersichtlich gegliedert und enthält alle Informationen, die für die praktische Arbeit am Computer notwendig sind: BASIC-Kurs mit Beispielen, Strukturiertes Programmieren, Dateiverwaltung, Grafikprogrammierung, Tips & Tricks.

Best.-Nr. MT 90385, ISBN 3-89090-385-1
(sFr. 35,90/öS 304,20) **DM 39,-**

COMMODORE 64

F. Ende

Das große Spielebuch - Commodore 64

1984, 141 Seiten

46 Spielprogramme · Wissenswertes über Programmiertechnik · praxisnahe Hinweise zur Grafikherstellung · alles über Joystick- und Paddleansteuerung · das Spielebuch mit Lerneffekt.

Best.-Nr. MT 603, ISBN 3-922120-63-6
(sFr. 27,50/öS 232,40) **DM 29,80**
Best.-Nr. MT 604 (Beispiele auf Diskette)
(sFr. 38,-/öS 342,-) **DM 38,-***
* inkl. MwSt. Unverbindliche Preisempfehlung.

S. Krute

Grafik & Musik auf dem Commodore 64

1984, 336 Seiten

68 gut strukturierte und kommentierte Beispielprogramme zur Erzeugung von Sprites und Klangeffekten · Sprite-Tricks · Zeichengrafik · hochauflösende Grafik · Musik nach Noten · spezielle Klangeffekte · Ton und Grafik · für fortgeschrittene Anfänger, die alle Möglichkeiten des C64 ausnutzen wollen.

Best.-Nr. MT 743, ISBN 3-89090-033-X
(sFr. 35,-/öS 296,40) **DM 38,-**

H. L. Schneider/W. Eberl

Das C64-Profihandbuch

1985, 413 Seiten

Ein Buch, das alle wichtigen Informationen für professionelle Anwendungen mit dem C64 enthält. Mit allgemeinen Algorithmen, die auch auf andere Rechner übertragbar sind, und vielen Utilities, getrennt nach BASIC- und Maschinenprogrammen. Besonders nützlich: erweiterte PEEK- und POKE-Funktionen.

Best.-Nr. MT 749, ISBN 3-89090-110-7
(sFr. 47,80/öS 405,60) **DM 52,-**

W.-J. Becker/M. Folprecht

Programmieren unter CP/M mit dem C64

1985, 290 Seiten

Wenn Sie wissen wollen, wie das Betriebssystem CP/M-2.2 auf dem C64 implementiert ist, außerdem einiges über Turbo-Pascal, Nevada-Fortran, MBASIC-80 erfahren wollen, dann ist dieses Buch genau richtig für Sie! Mit Schaltplänen zur eigenen Fertigung des CP/M-Moduls. Für eingefleischte C64-Profis.

Best.-Nr. MT 751, ISBN 3-89090-091-7
(sFr. 47,80/öS 405,60) **DM 52,-**

J. Mihalik

35 ausgesuchte Spiele für Ihren Commodore 64

1984, 141 Seiten

Programmieren Sie selbst 35 faszinierende Spiele · geschrieben in Commodore-64-BASIC · mit Farbe, Grafiken und Ton · Vorschläge zur Programmabwandlung · für kreative Computerfans, die ihre Programmierkenntnisse vertiefen wollen!

Best.-Nr. MT 774, ISBN 3-89090-064-X
(sFr. 23,-/öS 193,40) **DM 24,80**

W. Kassera/F. Kassera

C64 - Programmieren in Maschinensprache

1985, 327 Seiten inklusive Beispieldiskette

In diesem Buch finden Sie über 100 Beispiele zur Assembler-Programmierung mit viel Kommentar und Hintergrundinformationen: Das Schreiben von Maschinenprogrammen · Rechnen und Texten mit vorhandenen Routinen · Bedienung von Drucker und Floppy · Wie man BASIC- und Maschinenprogramme verknüpft · Erstellen von eigenen Befehlen in Modulform. Für Profis!

Best.-Nr. MT 830, ISBN 3-89090-168-9
(sFr. 47,80/öS 405,60) **DM 52,-**

P. W. Dennis G. Minter

Spiele für den Commodore 64

1984, 196 Seiten

Bewährte alte und raffinierte neue Spiele für Ihren Commodore 64 · klar und übersichtlich gegliederte Programme im Commodore-BASIC · Sie lernen: wie man Unterprogramme einsetzt · eine Tabelle aufbauen und verarbeiten · Programme testen · mit vielen Programmiertricks · für Anfänger.

Best.-Nr. MT 90074, ISBN 3-89090-074-7
(sFr. 23,-/öS 193,40) **DM 24,80**
Best.-Nr. MT 795 (Beispiele auf Diskette)
(sFr. 38,-/öS 342,-) **DM 38,-***
* inkl. MwSt. Unverbindliche Preisempfehlung.

K. Schramm

Die Floppy 1541

1985, 434 Seiten

Für alle Programmierer, die mehr über ihre VC-1541-Floppystation erfahren wollen. Der Vorgang des Formatierens · das Schreiben von Files auf Diskette · die Funktionsweise von schnellen Kopier- und Ladeprogrammen · viele fertige Programme · Lesen und Beschreiben von defekten Disketten · Für Einsteiger und für fortgeschrittene Maschinensprache-Programmierer.

Best.-Nr. MT 90098, ISBN 3-89090-098-4
(sFr. 45,10/öS 382,20) **DM 49,-**
Best.-Nr. MT 710 (Beispiele auf Diskette)
(sFr. 29,90/öS 269,10) **DM 29,90***
* inkl. MwSt. Unverbindliche Preisempfehlung.

Die angegebenen Preise sind Ladenpreise

607326/1

Sie erhalten Markt & Technik-Bücher bei Ihrem Buchhändler
Markt & Technik Verlag AG, Unternehmensbereich Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München

C64 - Programmieren in Maschinensprache

Wer schon einige Zeit in BASIC programmiert hat, wird bald an die Grenzen dieser Computersprache stoßen: Rechnen, Auswerten, Datenübertragungen, Simulationen, das alles läuft manchmal unerträglich langsam ab, und eine ganze Reihe von Problemen ist daher aus Zeitgründen nicht oder nur unbefriedigend lösbar.

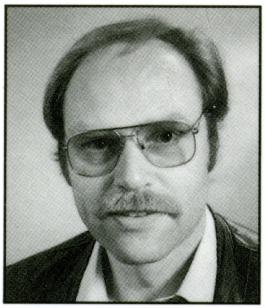
Auch mit anderen Sprachen gibt es diese Schwierigkeiten. Warum also nicht gleich in ASSEMBLER arbeiten, das den Einsatz der Maschinensprache ermöglicht, mit der direkt auf das Herz des Computers zugegriffen werden kann?

Hat man sich erst einmal mit ASSEMBLER vertraut gemacht, dann stehen auch die betriebsinternen ROM-Routinen zur Verfügung, die sich rasch und effektiv einsetzen lassen, wenn man weiß, wo sie liegen und wie sie vorbereitet werden müssen.

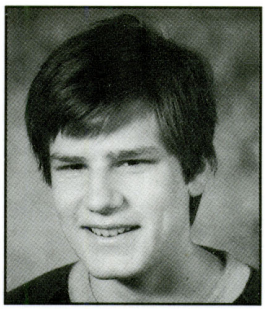
Wir zeigen in diesem Buch den COMMODORE C 64-, den 40er- und den 80er-Besitzern in weit über 100 ASSEMBLER-Beispielen mit viel Kommentar und Hintergrundinformation

- wie man Maschinenprogramme schreibt
- wie man mit vorhandenen Routinen rechnet und textet
- wie man Drucker und Floppy bedient
- wie man BASIC- und Maschinenprogramme verknüpft
- wie man eigene Befehle in Modulform erstellt.

Sie brauchen dazu nicht stundenlang Ihr ROM-Listing nach den passenden Einsprüngen und Adressen zu durchsuchen. Das haben wir hier für Sie erledigt.
Wenn Sie ernsthaft in ASSEMBLER programmieren wollen, finden Sie für Ihr COMMODORE-Gerät mit diesem Buch das richtige Nachschlagewerk.



WINFRIED KASSERA
Jahrgang 1942, verheiratet, zwei Söhne. Lehrer an der Realschule Neu-Ulm. Ab 1982 Einführung von COMMODORE-Geräten für das Fach Informatik. Dadurch intensive Beschäftigung mit diesen Rechnern. In der Freizeit leidenschaftlicher Flieger mit Lizenzen für Motor- und Segelflugzeuge. Seit 1966 auch Fluglehrer und später Prüfer. Für den Einsatz in der Ausbildung der Luftfahrer Entwicklung von Simulationsprogrammen als Maschinenprogramme für die 65er-Prozessoren. Lehrbuch für Segelflieger »Flug ohne Motor« bereits in der 8. Auflage. Tätigkeiten als Übersetzer und Lektor für Luftfahrtbücher.



FRANK KASSERA
Jahrgang 1969, Geburtsort Ulm. Grundschulbesuch von 1975 bis 1979. Eintritt in das Illertal-Gymnasium von Vöhringen 1979. Mehrfache Auszeichnungen als Klassen- und Schulbester. 1981 Einstieg in die Computerwelt, zunächst in BASIC, dann in ASSEMBLER. Besonderer Schulverdienst beim Aufbau des Faches Informatik. Mitarbeit bei verschiedenen Programmentwicklungen in BASIC und ASSEMBLER für COMMODORE-Systeme. Umsetzung von 40/80er-Maschinenprogrammen auf den C 64. Weitere Hobbies: Musik, Schwimmen.

ISB N 3-89090-168-9

Markt&Technik

4 001057 901681

DM 52,-
sFr. 47,80
öS 405,60